Programming Methodology

for

CDC® CYBER 205 Vector Processor

H. J. Rothmund
K. L. Murphy

CONTROL DATA CORPORATION

27 August 1980

1.0 INTRODUCTION

The CONTROL DATA® CYBER 205 Computer System is based on the architecture of the CYBER 203 and STAR 100 with significant logical and electronic advances over those systems. The CYBER 205 uses a second generation vector processor that makes use of the latest technology in LSI logic. Array and matrix operations can now be performed at rates up to eight times faster than those of the CYBER 203 and STAR-100. The performance of the data motion instructions (gather, scatter, compress, etc.) has been improved up to 32 times faster than those of the previous systems.

The CYBER 205 is designed to perform both conventional and special arithmetic used in the simulation of complex physical environments, which could previously not be performed, because no computer system was capable of delivering the required computational power within a reasonable time frame.

This paper illustrates some of the CYBER 205 hardware capabilities and provides an introduction to the programming methodology for the CYBER 205 vector processor.

A summary of the architecture of the Control Data CYBER 205 computer is given. The CYBER 205 is a superscale, highspeed scientific computer system with segmented scalar functional units, a vector processor containing up to four floating-point pipelines and up to four million 64-bit words of semiconductor memory. Peak performance on the vector processor is 800 million 32-bit floating point operations per second for linked multiply and add triads. Figure 2-1 is a diagram of the CYBER 205.

1 Central Processor

The central processor unit consists of three functional areas:

- Scalar processor
- Vector processor
- Input/output

The scalar processor decodes all instructions from central memory and directs vector/string instructions to the vector processor for execution. An instruction stack provides buffering for 64 virtually addressed 64-bit words, which can contain up to 128 32-bit instructions, 64 64-bit instructions, or a combination of both. The processor is capable of issuing instructions at a peak rate of one instruction every 20 nanoseconds, provided there are no conflicts. The scalar processor can execute scalar instructions in parallel with most vector instructions if there are no memory references generated by the scalar instruction for operands. To minimize memory references by scalar instructions, a register file consisting of 256 general purpose registers is used to maintain single element operands within subprograms.

The scalar processor contains five independent functional units:

- Add/Subtract unit
- Multiply unit
- Logical unit
- Single cycle unit
- Divide/Square Root/ Convert unit

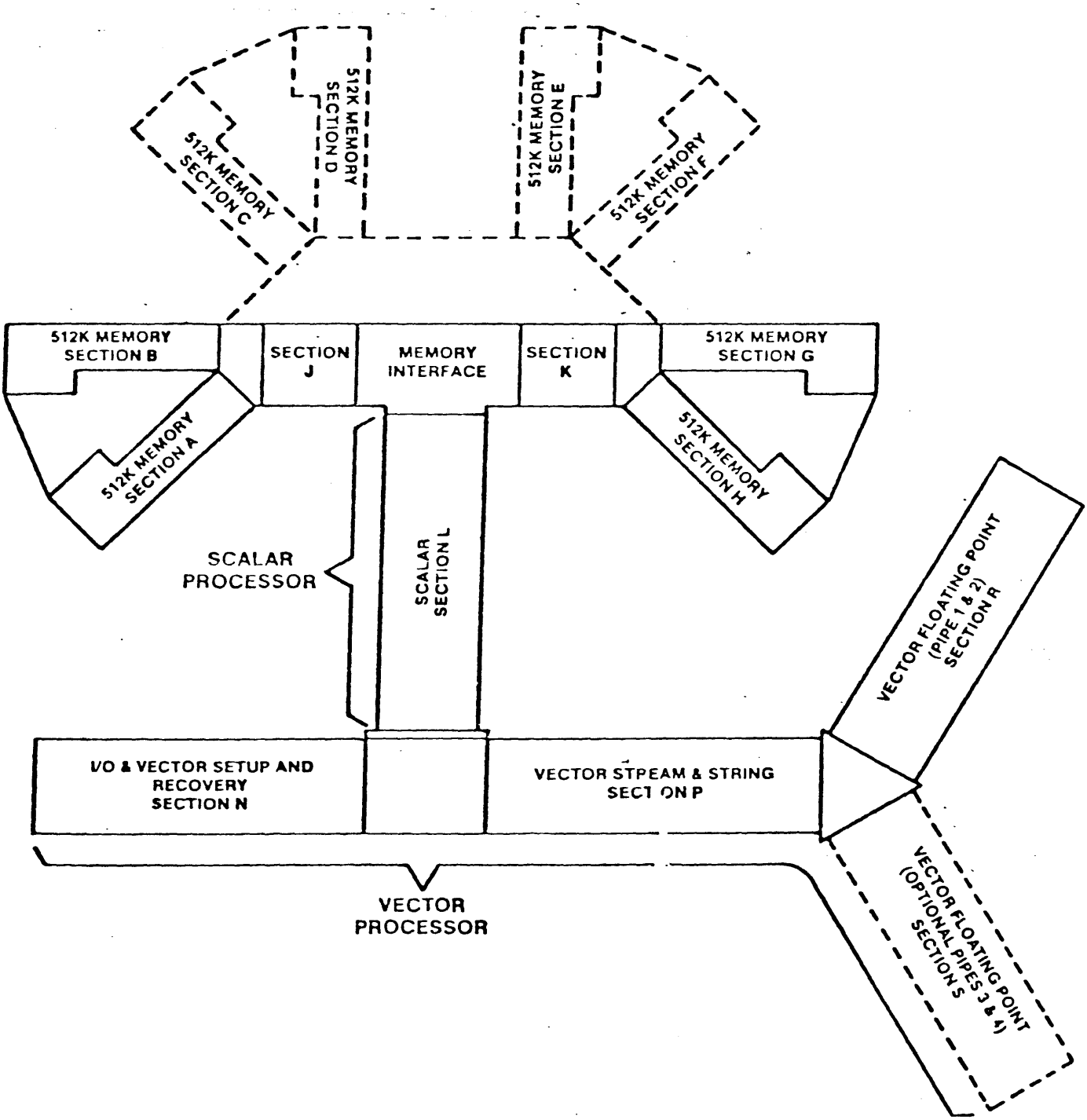

512K MEMORY SECTION C
512K MEMORY SECTION D
512K MEMORY SECTION E
512K MEMORY SECTION F
512K MEMORY SECTION B
SECTION J
MEMORY INTERFACE
SECTION K
512K MEMORY SECTION G
512K MEMORY SECTION A
512K MEMORY SECTION H
SCALAR PROCESSOR
SCALAR SECTION L
VECTOR FLOATING POINT (PIPE 1 & 2) SECTION R
I/O & VECTOR SETUP AND RECOVERY SECTION N
VECTOR STREAM & STRING SECTION P
VECTOR FLOATING POINT (OPTIONAL PIPES 3 & 4) SECTION S
VECTOR PROCESSOR

Figure 2-1. CYBER 205

All functional units are segmented and capable of accepting new operands every 20 nanoseconds with the exception of the Divide/Square Root/Convert unit which must complete each operation before a new one can begin. All units are capable of being shortstopped. Shortstop is a process by which a result from an arithmetic unit can be returned directly to either input of any arithmetic unit without waiting for the result to be stored in the register file.

Figure 2-2 is a diagram of the functional components of the scalar processor. Peak performance of the scalar processor is 50 million 32-bit or 64-bit floating point operations per second.

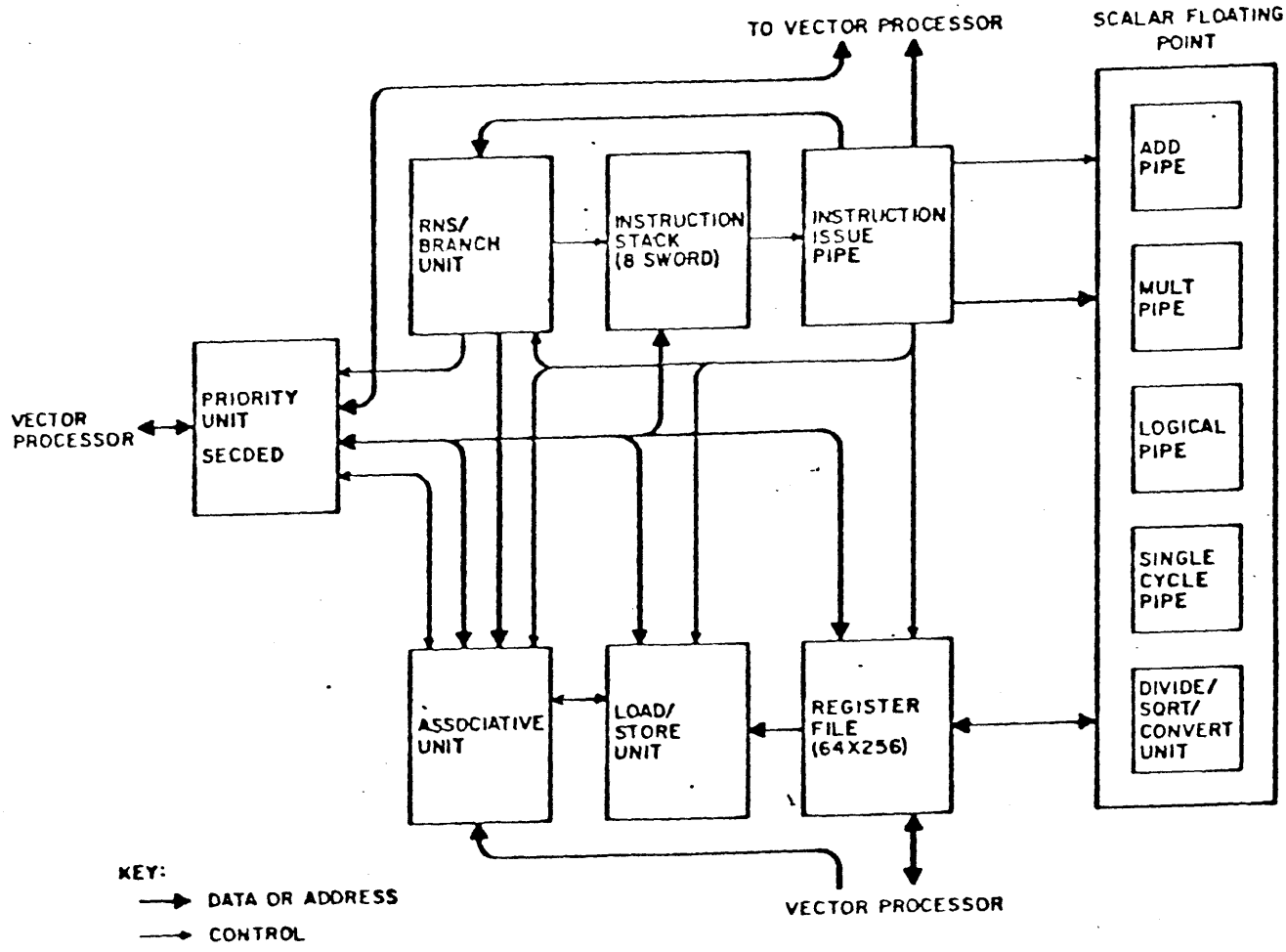


Figure 2-2. Functional Components of the Scalar Processor

The vector unit consists of the stream unit, the string unit, and the segmented vector pipeline units.

The stream unit receives decoded instructions from the scalar unit and controls the data streams between central memory and the vector pipelines.

The string unit performs operations on byte and bit vectors which are used for logical and control vector operations.

The vector pipeline units are used for vector add/subtract, multiply, and divide/square root operations. For vector addition, subtraction, and multiplication, the computer contains one, two, or four 64-bit pipelines. The vector floating point arithmetic instructions not only perform the defined arithmetic operation, but can perform additional operations without any increase in operation time. These additional operations are:

- The magnitude (absolute value) of the operands from one or both input vectors are used.

- The coefficients of the operands from one input vector are complemented before they are used.

- The coefficients of all positive operands from one input vector are made negative before they are used.

Table 2-1 provides approximate vector timing information for selected CYBER 205 instructions based on 64-bit operands. Instructions using 32-bit operands or linked triadic operations run at twice the speed; linked 32-bit triadic operations run at four times the speed.

TABLE 2-1. APPROXIMATE TIMING CHART FOR SELECTED CYBER 205 INSTRUCTIONS

Let N = number of output operands (64-bit)
Let Z = length of control vector

Vector Operation	Cycles (20 nsec) 2 Pipes 32-Bit	2 Pipes 64-Bit	4 Pipes 32-Bit	4 Pipes 64-Bit
Add/Subtract	51 + N/4	51 + N/2	51 + N/8	51 + N/4
Multiply	52 + N/4	52 + N/2	52 + N/8	52 + N/4
Linked Multiply and Add	84 + N/4	84 + N/2	84 + N/8	84 + N/4
Divide/Square root	80 + N/.61	80 + N/.32	80 + N/1.22	80 + N/.64
Divide/Square root Upgrade	80 + N/1.22	80 + N/.64	80 + N/2.44	80 + N/1.28
Periodic Gather		39 + N/.8		39 + N/.8
Periodic Scatter		71 + N/.8		71 + N/.8
Compress		52 + Z/2		52 + Z/4
Merge		58 + Z/2		58 + Z/4

Therefore the peak performance on linked triads using 32-bit operands is 800 million floating point operations (addition, subtraction, and multiplication) per second (MFLOPS) for a computer with four pipelines and 400 MFLOPS for 64-bit operands with four pipelines.

The input/output system contains 8 or 16 I/O ports, each 32-bits in width. Each I/O port is capable of transferring up to 200 million bits per second. Total bandwidth for the input/output system is 3200 million bits per second.

2.2 Memory

The memory subsystem is available in sizes of 1 million, 2 million, or 4 million 64-bit words. Memory words are 78 bits long, providing a 64-bit data word and 14 bits for single error correction and double error detection (SECDED); seven bits for each 32-bit half word. Each half million words of memory contain 16 memory stacks arranged in eight phased banks. Sequential addresses are assigned to different banks by using bank phasing. Because the banks are independent, a bank can begin a memory cycle before adjacent banks have completed previously initiated cycles.

Memory is addressable by single bits, 32-bit half words, 8-bit bytes, or 64-bit full words.

Memory bandwidth of the CYBER 205 permits maximum transfer rate concurrently for all I/O ports, while supporting maximum vector processing rates.

3.0 VECTOR PROCESSING METHODOLOGY

Two phases exist during execution of a CYBER 205 vector instruction. The first phase is called the startup phase. During this phase the hardware sets up sufficient buffering for the data streams to ensure that no memory conflicts occur during the stream phase. Also the segmented functional units (pipes) are initially filled and emptied. The time consumed by the startup phase varies with the different vector operations (multiply, divide, etc.) but is independent of the vector length. The second phase is called the stream phase. The time spent during this phase is directly proportional to the vector length, mainly the number of cycles needed to produce one result times the number of result operands.

The performance of a vector operation is monotonic as a function of vector length. With increasing vector length the startup time becomes less significant and the performance becomes asymptotic to the peak performance rate. Figure 3-1 and tables 3-1 through 3-5 illustrate the performance in millions of floating point operations per second (MFLOPS) as a function of the vector length.

Before we can talk about vector processing on the CYBER 205, we have to define a CYBER 205 vector. A CYBER 205 vector is a set of contiguous memory locations. For real or integer vectors, the memory locations are 64-bit words or 32-bit half words; and for bit vectors they are bits.

An example of a vector is a one-dimensional FORTRAN array:

A (1)	A (2)		A (N)

Therefore, the FORTRAN DO loop:

```
      DO 1000  I = 1,N
      C (I) = A (I) + B (I)
1000  CONTINUE
```

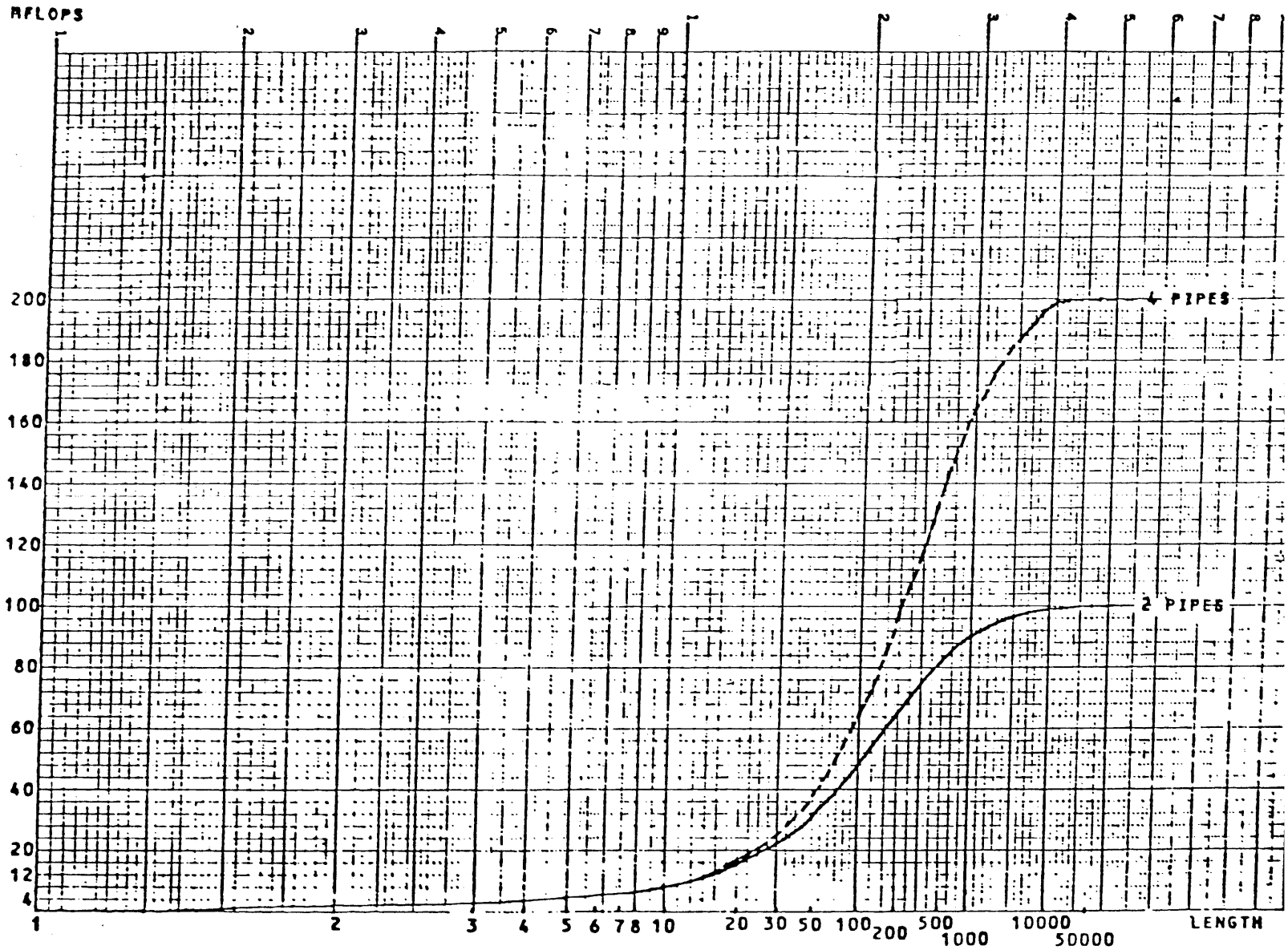


Figure 3-1. Approximate MFLOPS Rate as a Function of Vector Length for Vector Multiply With 64-bit Operands

becomes a vector addition, and the DO loop collapses into one instruction on the CYBER 205.

For the following DO loop:

```
      DO 1000 I = 1,N,M
      C (I) = A (I) * B (I)
1000  CONTINUE
```

If M = 1, this is a vector operation where vectors A, B, and C are contiguous sets of memory.

If M > 1, vectors A, B, and C are noncontiguous and therefore do not satisfy the definition of a CYBER 205 vector.

The case of M > 1, however, occurs frequently in general scientific and engineering applications; therefore, the CYBER 205 hardware provides a set of instructions to make noncontiguous data structures contiguous. These data motion instructions allow the generation of temporary contiguous vectors from noncontiguous data structures. In the following paragraphs we discuss some of the vectorization methods for noncontiguous data structures.

3.1 Control Store

Any vector operation associated with a control store uses a bit vector. A bit vector is defined as a CYBER 205 vector whose elements consist of a contiguous set of bits. Each bit corresponds to an element in a data vector. The control store operation stores the elements of a data vector whose corresponding bits in the bit vector are 1 into the corresponding positions of a result vector, and leaves the elements untouched whose corresponding bits are 0. The control store operation is done concurrently with the vector arithmetic operation and does not take additional computer time. The following example illustrates a control store:

```
      M = 3
      DO 1000 I = 1,N,M
      C(I) = A(I) - B(I)
1000  CONTINUE
```

It can be seen from this DO loop that the difference between A(I) and B(I) is calculated for every third element. By using the control store technique we calculate the difference for every element but store only every third result into C(I). To be able to do that we have to define a bit vector where every third bit is 1 and the remainder of the bits are 0.

bit vector -----> 1001001001001.....

Note that, independent of M, N arithmetic operations will be done. Thus, for M = 3, the peak result rate will be one-third the peak hardware rate.

The number of redundant operations is a linear function of M and efficiency decreases as M increases. Therefore, alternatives to the control store algorithm have to be used for larger values of M, which makes the control store an unattractive solution.

3.2 Vector Compress/Merge

The compress instruction will cause those elements of a data vector that correspond to 1's in the bit vector to be compressed into a temporary vector whose length is equal to the number of 1's in the bit vector.

Using the vector compress method on our previous example, we have to execute the following instruction sequence:

compress vector A into A_T
compress vector B into B_T
$C_T = A_T - B_T$
merge vector C with vector C_T

Each one of these steps corresponds to a CYBER 205 machine instruction. C_T, A_T, and B_T are contiguous temporary vectors of length $1 + (N-1)/M$.

TABLE 3-1. APPROXIMATE VECTOR ADD PERFORMANCE IN MFLOPS

N	1 Vector Pipeline		2 Vector Pipelines		4 Vector Pipelines	
	32-Bit	64-Bit	32-Bit	64-Bit	32-Bit	64-Bit
25	19.8	16.4	22.7	19.8	23.1	22.7
50	32.9	24.8	39.7	32.9	43.9	39.7
100	49.5	33.1	65.8	49.5	79.4	65.8
500	83.1	45.4	142.0	83.1	221.2	142.0
1,000	90.7	47.6	166.1	90.7	284.1	166.1
10,000	99.0	49.7	196.0	99.0	384.3	196.0
50,000	99.8	49.9	199.2	99.8	396.8	199.2
Asymptote	100.0	50.0	200.0	100.0	400.0	200.0

TABLE 3-2. APPROXIMATE VECTOR MULTIPLY PERFORMANCE IN MFLOPS

N	1 Vector Pipeline		2 Vector Pipelines		4 Vector Pipelines	
	32-Bit	64-Bit	32-Bit	64-Bit	32-Bit	64-Bit
25	19.5	16.2	21.6	19.5	22.7	21.6
50	32.5	24.5	39.1	32.5	43.1	39.1
100	49.0	32.9	64.9	49.0	78.1	64.9
500	82.8	45.3	141.2	82.8	219.3	141.2
1,000	90.6	47.5	165.6	90.6	281.5	165.6
10,000	99.0	49.7	195.9	99.0	384.0	195.9
50,000	99.8	49.9	199.2	99.8	396.7	199.2
Asymptote	100.0	50.0	200.0	100.0	400.0	200.0

TABLE 3-3. APPROXIMATE LINKED VECTOR MULTIPLY AND ADD PERFORMANCE IN MFLOPS

	1 Vector Pipeline		2 Vector Pipelines		4 Vector Pipelines	
N	32-Bit	64-Bit	32-Bit	64-Bit	32-Bit	64-Bit
25	26.0	22.9	27.8	26.0	28.7	27.8
50	45.9	37.3	52.1	45.9	55.6	52.1
100	74.6	54.3	91.7	74.6	104.2	91.7
500	149.7	85.6	239.2	149.7	342.5	239.2
1,000	171.2	92.3	299.4	171.2	478.5	299.4
10,000	196.7	99.2	387.0	196.7	749.6	387.0
50,000	199.3	99.8	397.3	199.3	789.4	397.3
Asymptote	200.0	100.0	400.0	200.0	800.0	400.0

TABLE 3-4. APPROXIMATE VECTOR DIVIDE AND SQUARE ROOT PERFORMANCE IN MFLOPS

	1 Vector Pipeline		2 Vector Pipelines		4 Vector Pipelines	
N	32-Bit	64-Bit	32-Bit	64-Bit	32-Bit	64-Bit
25	7.9	5.3	10.3	7.9	12.5	10.5
50	10.6	6.4	15.5	10.6	20.8	15.8
100	12.8	7.1	20.6	12.6	31.1	21.2
500	15.2	7.8	27.8	15.2	51.1	29.0
1,000	15.6	7.9	29.1	15.6	55.6	30.5
10,000	16.0	8.0	30.4	15.9	60.4	31.8
50,000	16.0	8.0	30.5	16.0	60.9	31.9
Asymptote	16.0	8.0	30.5	16.0	61.0	32.0

TABLE 3-5. APPROXIMATE VECTOR DIVIDE AND SQUARE ROOT UPGRADE PERFORMANCE IN MFLOPS

N	1 Vector Pipeline	2 Vector Pipelines		4 Vector Pipelines	
		32-Bit	64-Bit	32-Bit	64-Bit
25	(not available)	12.5	10.5	13.9	12.6
50		20.8	15.8	25.0	21.0
100		31.1	21.2	41.7	31.6
500		51.1	29.0	88.0	53.2
1,000		55.6	30.5	102.2	58.1
10,000		60.4	31.8	119.7	63.4
50,000		60.9	32.0	121.5	63.9
Asymptote		60.9	32.0	121.6	63.9

3.3 Periodic Gather/Scatter

The periodic gather instruction creates a temporary contiguous data vector by transmitting elements from another vector of the same data type indexed by a periodic index "M." The periodic scatter instruction is the inverse of the gather instruction.

Using the gather/scatter method as our example, the following instruction sequence has to be generated:

gather vector A into A_T
gather vector B into B_T
$C_T = A_T - B_T$
scatter vector C_T into vector C

C_T, A_T, and B_T are contiguous temporary vectors.

Figures 3-2 through 3-4 show the performance results measured in MFLOPS of the discussed vectorization methods for noncontiguous vectors. As it can be seen from those graphs, the best method chosen depends upon the values of M, N, and OPS (number of vector operations). In general, for small M's and OPS's, the increased number of redundant operations in the control store method reduces the performance. The compress/merge and gather/scatter methods result in a better performance in this case. With the increase of the number of vector operations, the time spent for the data motion operations (compress, merge, gather, and scatter) in comparison to the time spent for the vector operations becomes increasingly less significant. Also, with increasing periodic indexes M, the gather/scatter method clearly outperforms the other two methods. The periodic gather/scatter method is applied by the automatic vectorizer of the CYBER 205 FORTRAN compiler for periodic noncontiguous data structures.

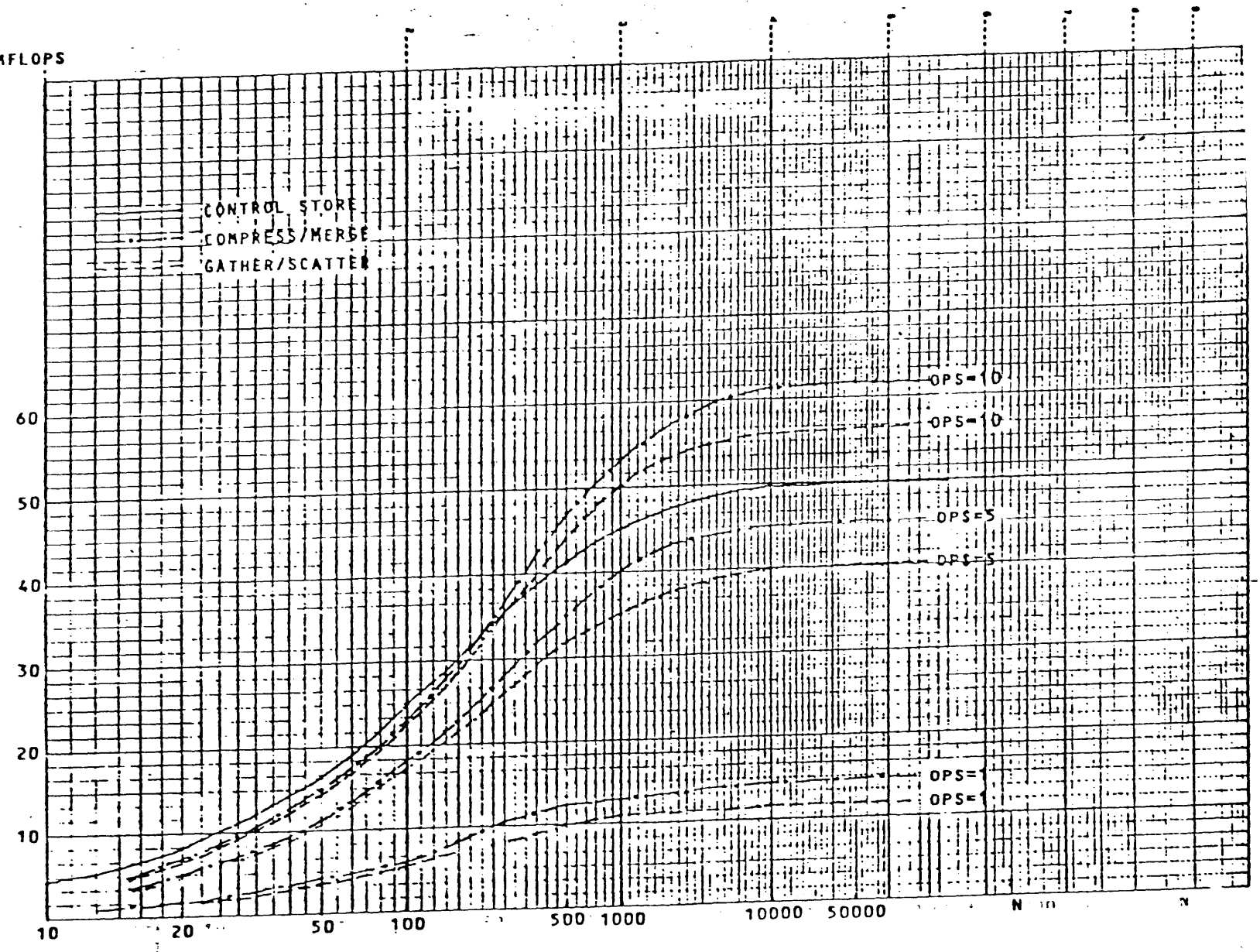


Figure 3-2. Approximate MFLOPS Rates for M = 2

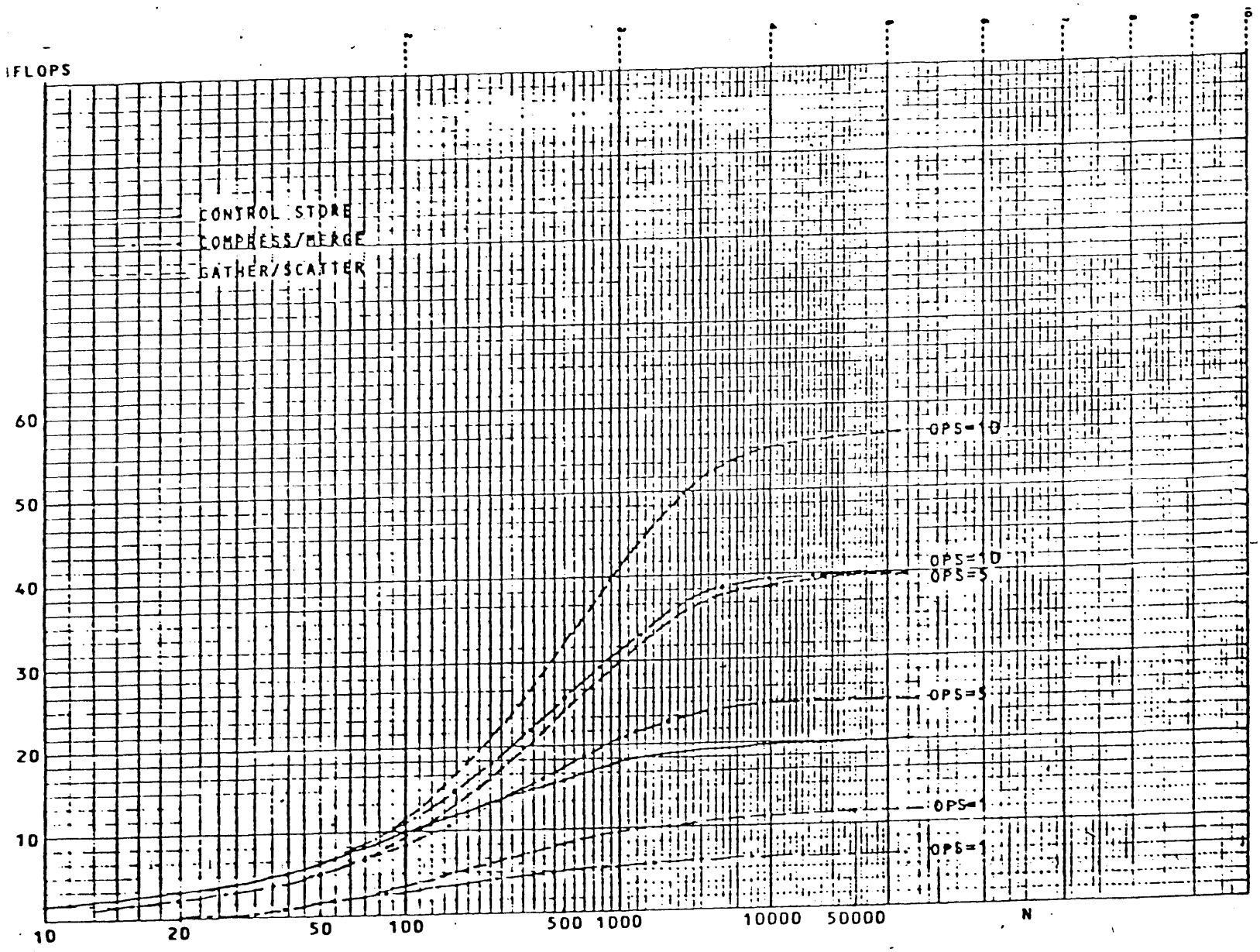


Figure 3-3. Approximate MFLOPS Rates for M = 5

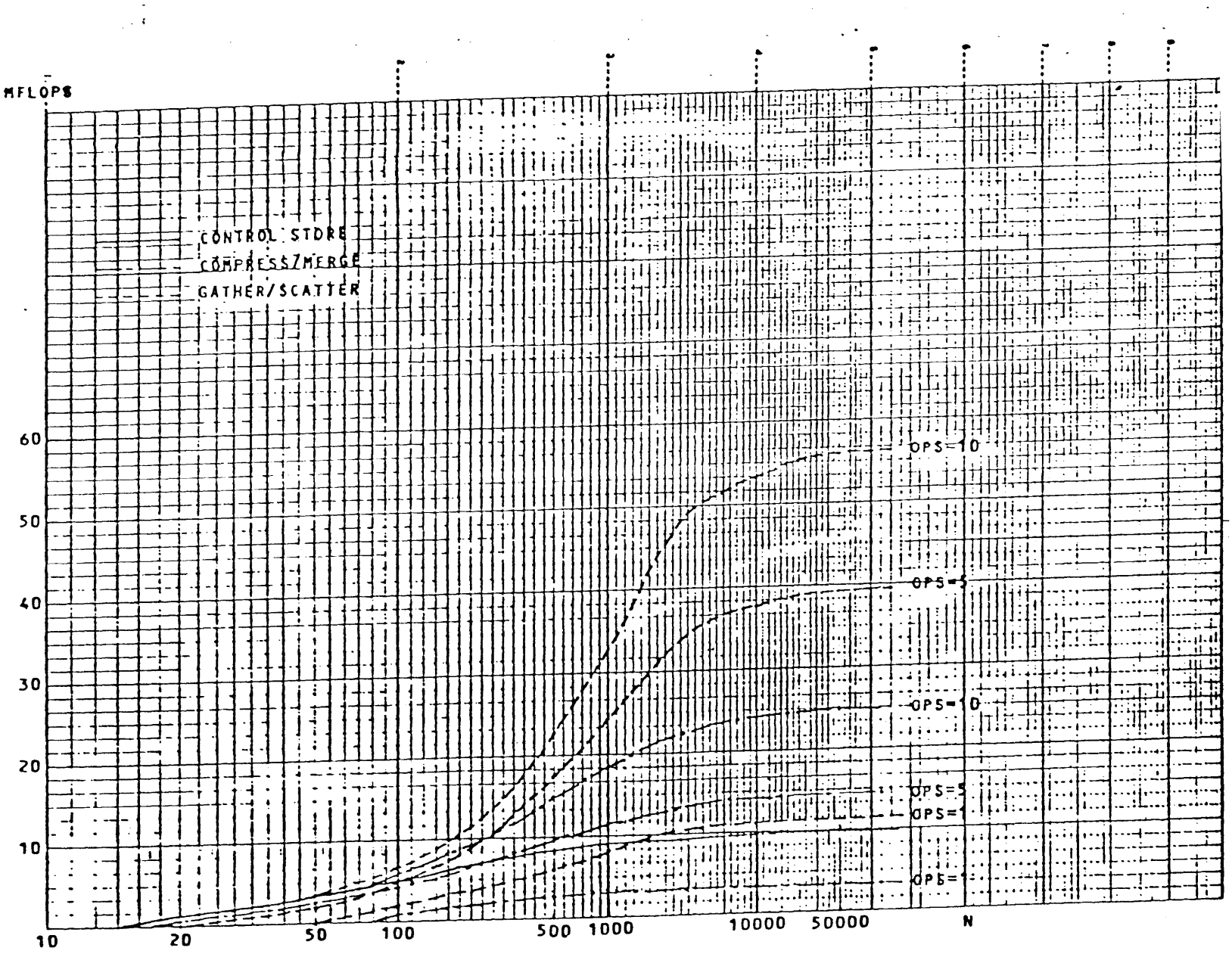


Figure 3-4. Approximate MFLOPS Rates for M = 10

3.4 Random Gather/Scatter

The random gather instruction uses the elements of an integer index vector as indices to take possibly disparate elements from a source data vector and make them contiguous in a temporary vector.

For an example, the following FORTRAN DO loop can be replaced by one gather instruction:

```
      DO 1000 J = 1,N
      B(J) = A(I(J))
1000 CONTINUE
```

The random scatter instruction is the inverse of the gather instruction.

```
      DO 1000 J = 1,N
      A(I(J)) = B(J)
1000 CONTINUE
```

4.0 MATRIX ADDITION AND MULTIPLICATION

For two full matrices stored in the same order as contiguous data structures, the matrix sum can be evaluated with one instruction, provided the size is 255 x 255 or less.

For the product of two N x N matrices, the inner (or DOT) product of the jth row of one matrix and the kth column of the other matrix is formed. If one matrix is stored by rows and the other one by columns, the product matrix can be formed with N^2 inner products.

The timing for the inner product instruction is:

$$107 + N \text{ cycles} \quad \text{(each cycle is 20 nsec)}$$

Therefore, the total time is:

$$107N^2 + N^3$$

If the two matrices are not stored by rows and columns, one must gather the appropriate entities. However, there is an algorithm that does not require the gather and is asymptotically four times faster than the inner product algorithm. This algorithm is called the "outer product" and is based on the fact that:

$$\begin{pmatrix} C_{1k} \\ \cdot \\ \cdot \\ \cdot \\ C_{nk} \end{pmatrix} = B_{1k} \begin{pmatrix} A_{11} \\ \cdot \\ \cdot \\ \cdot \\ A_{n1} \end{pmatrix} + B_{2k} \begin{pmatrix} A_{12} \\ \cdot \\ \cdot \\ \cdot \\ A_{n2} \end{pmatrix} + \cdots + B_{nk} \begin{pmatrix} A_{1n} \\ \cdot \\ \cdot \\ \cdot \\ A_{nn} \end{pmatrix}$$

If A is stored by columns and B is stored in any fashion, then:

```
      DO 1000 K = 1,N
      C(1,K,N) = B(1,K) * A(1,1;N)
      DO 1000 L = 2,N
      C(1,K;N) = C(1,K;N) + B(L,K) * A(1,L;N)
1000  CONTINUE
```

will compute matrix C and store it by columns.

The time for this algorithm is:

$N^3/2 + O(N^2)$ on a 2-pipe machine
$N^3/4 + O(N^2)$ on a 4-pipe machine

Note that the inner loop of the algorithm is a linked triad. Not only is this algorithm asymptotically four times faster than the first algorithm, but the $O(N^2)$ is much smaller due to the fact that no data motion is required.

The following table shows the necessary data structures to achieve the matrix operations listed:

	FULL MATRIX ADDITION C = A + B	FULL MATRIX MULTIPLICATION C = AB
DATA STRUCTURE	A rows A columns B rows B columns C rows C columns	A columns A * B * B rows C columns C rows
MEAN VECTOR LENGTH	N^2	N
CYCLE COUNT		
2-pipe CYBER 205	$N^2/2 + O(1)$	$N^3/2 + O(N^2)$
4-pipe CYBER 205	$N^2/4 + O(1)$	$N^3/4 + O(N^2)$

Figure 4-1 shows the MFLOPS rates for a matrix multiplication using the "outer product" algorithm and assumes the total number of operations to be $2N^3$.

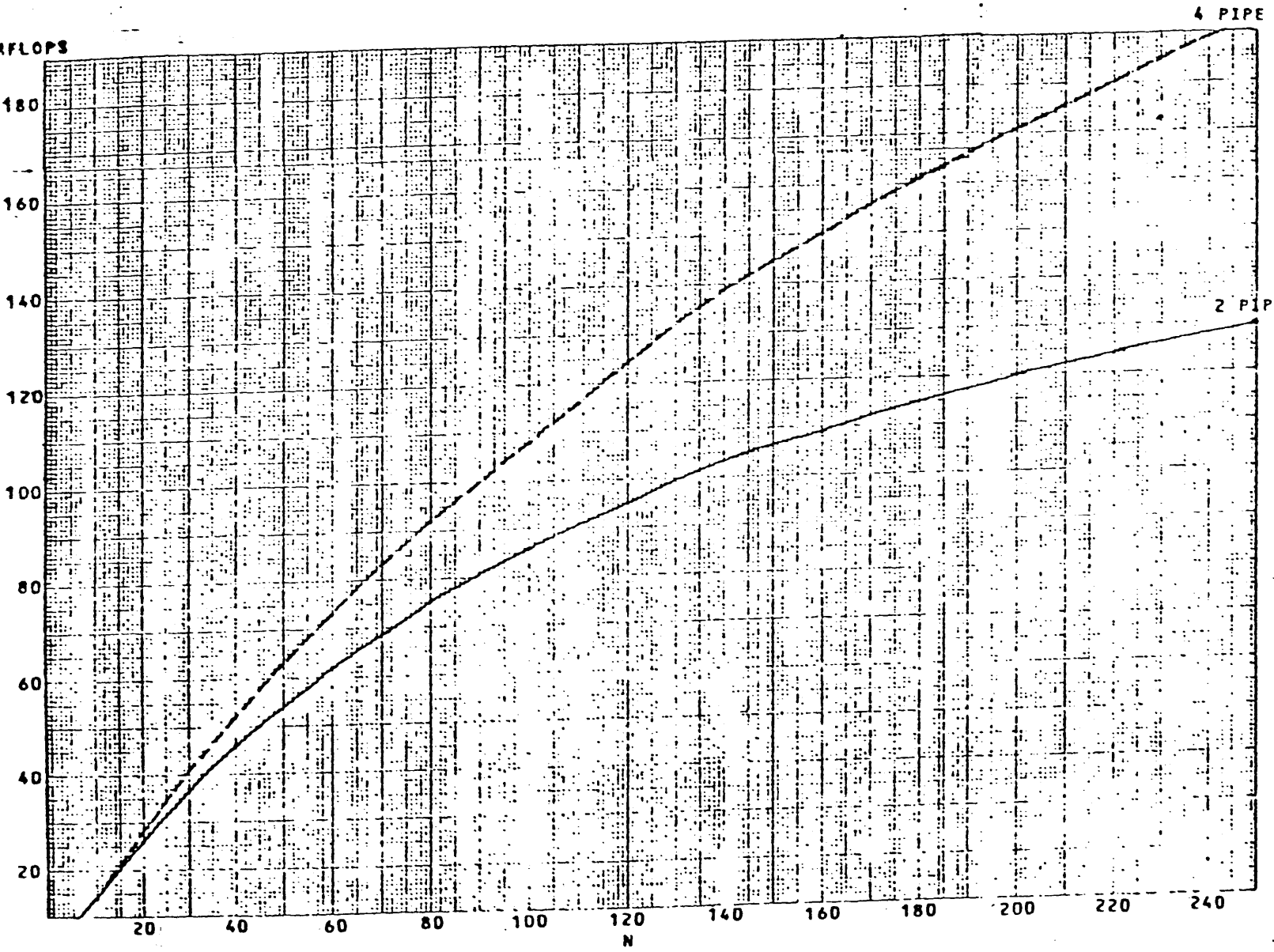


Figure 4-1. Approximate MFLOPS Rates of NxN Matrix Multiply (64-bit Operands)

5.0 CONCLUSION

The material presented in this paper is only an introduction to the capabilities of the CYBER 205. Many syntactic FORTRAN kernels are reduced to single instructions due to the vector processing capabilities of the system.

The concept of vector processing introduced in this paper shows that even with a vector length of less than 100 elements, the performance is very impressive. More importantly, implicit vectorization of FORTRAN kernels is done automatically by the compiler. However, to achieve peak performance rates and to make full use of the powerful CYBER 205 instruction set, vector numerical analysis type algorithms have to be developed.

BIBLIOGRAPHY

Control Data Corporation, CYBER 200 Hardware Manual, 60256010.

Control Data Corporation, CYBER 200 FORTRAN Language, 60457040.

Kascic, M. J., "Vector Processing on the CYBER 200."

Control Data Corporation, CYBER 200 Series Model 205 Computer System.

VECTOR PROCESSING ON THE CYBER 200

M. J. KASCIC, JR.

Control Data Corporation
4201 North Lexington Avenue
St. Paul, Minnesota 55112

First published in the Infotech State of the Art Report "Supercomputers", Infotech International Limited, Maidenhead, UK (1979).

VECTOR PROCESSING ON THE CYBER 200

M. J. KASCIC, JR.
Control Data Corporation
4201 North Lexington Avenue
St. Paul, Minnesota 55112

ABSTRACT

The CDC CYBER 200 series computer provides computational capability measured in the hundreds of mflops (millions of floating operations per second). This capability is due to its vector processing hardware.

The goal of this paper is twofold:

1. To illustrate this hardware capability by a selected set of syntactic kernels, and

2. To provide an introduction to the applicability of vector processing to some basic algorithms of numerical linear algebra.

1. INTRODUCTION

Although basic advances in technology have contributed significantly to the ability of successive generations of computers to solve problems faster, in the last decade the architectural revolution known as vector processing has been most responsible for pushing out the frontiers of calculational capability.

Rather than acting on individual operands, the vector processor acts on assemblages of operands (called by various authors vectors, arrays, strings, etc.). This allows the processor to configure itself more efficiently to perform a certain operation.

Let me illustrate with a trivial example. Suppose one thousand floating operands are to be added to another thousand floating operands and the results stored in a thousand words in memory. The traditional hardware (hereafter termed scalar hardware) must execute code somewhat similar if not identical to the following:

Load, Load, Add, Store, Index and Branch.

Thus, the operands must be fetched individually from memory into a staging area called the register file, then added, then stored back into memory one at a time. Finally, the decision must be made whether there are any more passes to be made through the code sequence. For this simple example, the scalar processor must execute five or six thousand instructions. Moreover, when executed in the order given above, much of the time various parts of the processor are idle, e.g., while waiting for operands to be loaded into the register file, the adder is not doing any useful work. While it is true that software can alleviate some of this problem by techniques such as bottom load - top store code generation, it is still true that the individual instructions perform such a small piece of the total algorithm that the processor can not "know" how best to overlap its various functional units. In particular, the branch instruction is usually several times slower than an arithmetic instruction. Recognizing this problem, crafty programmers have come up with the technique of loop unrolling. Essentially, this amounts to doing several pieces of arithmetic per branch. This not only cuts down on the number of branches, but allows a segmented arithmetic functional unit to keep more of its segments or stations busy during a given time period.

Several factors limit this technique in a scalar processor:

1. the size of the instruction stack which must hold all the instructions necessary for one pass.

2. the ability of the Load/Store unit to fetch and return operands to memory.

3. the number of registers to act as a staging area.

The ultimate evolution of this technique is to design hardware that

1. has a high bandwidth (ability to pass data) between the memory and the calculational unit to allow simultaneous Load/Store overlapped with calculation.

2. will route the proper number of operands without a need for branching as such or an explicit staging area in a register file.

This solution is embodied in the CYBER 200 vector processing hardware. The CYBER 200 series is an outgrowth of the STAR 100 processor. The reader that is interested in more hardware detail may consult (1), (2), and (5).

Basically, there is one hardware instruction to perform up to 65535 additions, subtractions, multiplications or divisions. At issue of such an instruction, the CYBER 200 "organizes" itself to carry out the one operation on many operands in the most efficient manner.

The execution of a CYBER 200 vector arithmetic instruction consists of two phases. The first phase is the startup phase. The time consumed by this phase is independent of vector length. During this period, sufficient buffering is automatically set up by the hardware to ensure no memory conflict among the various streams of data into and out of the memory. Also the general purpose segmented functional units (called pipes) are initially filled (and emptied). The second phase is the stream phase. The time consumed by this phase is directly proportional to the vector length, the constant of proportionality being the average number of cycles needed to produce one result operand.

The foregoing should make it clear that performance will be monotonic as a function of vector length since the longer the vector, the more time to amortize the startup. The asymptotic performance is that performance that would occur if there were no startup.

It should also be clear that the number of instructions issued in a given time period is no longer a valid measure of performance for a vector processor. Indeed, the simple addition kernel that we considered before can be effectively computed on the CYBER 200 with one instruction. A more useful measure of performance is the megaflop (mflop), the capability to perform a million floating operations in one second.

In terms of mflops, the state of the art for scalar computing is on the order of magnitude of one to ten mflops. By the very nature of scalar computing this number varies from kernel to kernel, as well as from programmer to programmer. By contrast, consider the following tables of vector performance of the CYBER 200 series. (The CYBER 203 numbers represent actual performance data. The CYBER 203E, a CYBER 203 with LSI vector pipes, is at the conclusion of the design phase at the time of this writing. Thus its performance represents expected design goals.) The tables give mflop rates for 64 bit arithmetic operations. For 32 bit arithmetic, the CYBER 203 add rate is doubled, the CYBER 203 multiply rate is quadrupled and all CYBER 203E rates are doubled. The bottom lines in each case are asymptotic performance.

(1.1)

CYBER 203

VECTOR LENGTH	ADDITION	MULIPLICATION
25	7.5	3.4
50	13.0	6.0
100	20.7	9.7
500	38.9	19.0
1000	43.8	21.6
10000	49.3	24.6
50000	49.9	24.9
	50.0	25.0

(1.2) CYBER 205 VECTOR ADD OR MULTIPLY MEGAFLOP RATE

VECTOR LENGTH	2 PIPE, 64-BIT	4 PIPE, 64-BIT 2 PIPE, 32-BIT	4 PIPE, 32-BIT
32	23.9	27.1	29.1
64	38.6	47.8	54.2
100	49.5	65.8	78.1
250	71.0	109.6	150.6
500	83.1	142.0	219.3
1000	90.7	166.1	284.1
10000	99.0	196.0	384.3
50000	99.8	199.2	396.8
	100.0	200.0	400.0

In addition, the CYBER 203E is expected to have linked triad capability as follows. Various triadic operations involve two input vector streams and one input scalar stream. Such triads will be calculated as one vector operation. Two important examples of such triads that we will see in the succeeding sections are

1. Vector + Scalar*Vector
2. (Vector + Scalar) *Vector

The CYBER 203E performance for 64 bit operands on such triads will be

(1.3) CYBER 205 LINKED TRIAD MEGAFLOP RATE

VECTOR LENGTH	2 PIPE, 64-BIT	4 PIPE, 64-BIT 2 PIPE, 32-BIT	4 PIPE, 32-BIT
32	31.7	34.4	36.0
64	54.7	63.4	68.8
100	74.1	90.9	102.0
250	119.0	168.9	213.7
500	149.3	238.1	337.8
1000	170.9	298.5	476.2
10000	196.7	386.8	749.1
50000	199.3	397.3	789.3
	200.0	400.0	800.0

Thus vector processing even now offers an order of magnitude performance over scalar and very shortly will offer two orders of magnitude. Indeed, processors are now being designed for the mid 80's with sustainable 1000 mflop performance and peak rates of 3000 mflops, a full three orders of magnitude over scalar performance.

One of the fundamental laws of the universe is that you do not get something for nothing. This law certainly holds true for vector processing. We shall see in the next two sections that some programming sophistication is needed to get the full several orders of magnitude performance. Of course, the same law holds true for scalar programming. There is a remarkable difference between the performance of "casual" FORTRAN and "tuned" code on such sophisticated scalar processors as the CDC 7600 (or the CYBER 200 for that matter). The difference is that the performance range on scalar processors is narrower, i.e., the best performance increase one can expect to get is usually a factor of two or three. Now we are talking about orders of magnitude. The stakes have gone up. Thus it is crucial to understand how to use the vector processing capability intelligently.

2. VECTOR PROCESSING

In the introduction we spoke of the programming discipline needed for vector processing. A large part of this discipline is the recognition of data structure. To be more precise, let us ask the question, "What is a vector on the CYBER 200?"

(2.1) **Def. Vector – Contiguous set of memory locations**

For real or integer vectors, the memory locations are words, for complex vectors pairs of words, for bit vectors they are bits.

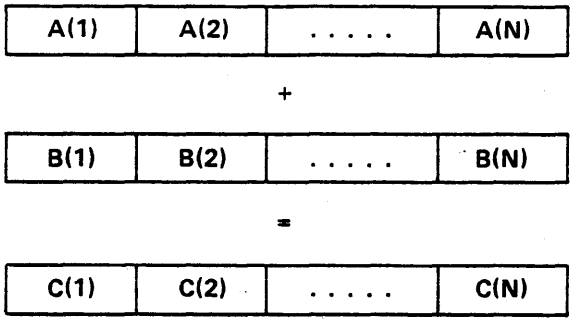
Up to this point we have been discussing the hardware of vectors. It is time to bring software into the picture. Where or how do vectors make their appearance from the programmers point of view?

The most primitive example of a vector is a one-dimensional FORTRAN array:

Thus, the fundamental FORTRAN DO loop

(2.2)
```
DO 9000 I = 1,N
C(I) = A(I) + B(I)
CONTINUE
```

actually performs the operation of a vector addition:



How does one code the vector addition?

There are two basic approaches to vectorization. One approach is so called automatic vectorization. This means writing traditional scalar FORTRAN and depending on a "vectorizer", a software module, to "see" the vector construct. The other approach is to provide high level language constructs that directly access the vector hardware.

It is not the purpose of this paper to debate the pros and cons of these two approaches. As any complete system should, the CYBER 200 has both options available.

The present task is to explore the second approach, with the explicit goal of delineating a wide range of CYBER 200 vector hardware primitives along with concomitant FORTRAN constructs.

The most basic software construct in the family that we shall persue is the descriptor.

(2.3) **Def. Descriptor — Pointer to a vector**

The internal format of a descriptor contains the starting address and the length of the vector pointed to. There are two FORTRAN syntaxes for the descriptor.

The explicit descriptor has the form

(2.4) **"array name"("index";"integer expression")**

which points to the vector whose first element is

(2.5) **"array name"("index")**

and whose length is

"integer expression".

Thus A(1;N) is an explicit descriptor pointing to the vector whose first element is A(1) and whose length is N.

There is also a variable type called descriptor which allows one to assemble the information for a descriptor and access it by name. Thus the declarative

(2.6)
```
DESCRIPTOR AD
```

informs the compiler that the variable name AD is to be a descriptor. The executable statement

(2.7)
```
ASSIGN AD,A(1;N)
```

compiles into code to form the descriptor and store it under the name AD. Since the ASSIGN statement is an executable statement, the vector pointed to by AD may be dynamically changed.

Further details about the FORTRAN descriptor construct may be found in (3) and (5).

One uses descriptors in artihmetic statements as follows:

(2.8)
```
descriptor = descriptor op descriptor
```

compiles into code to perform op on the vectors pointed to by the descriptors on the right hand side and store the result in the vector pointed to by the descriptor on the left hand side.

Bringing all of these facts together, the DO loop (2.2) reduces to

(2.9)
```
C(1;N) = A(1;N) + B(1;N)
```

or equivalently

(2.10)
```
CD = AD + BD
```

if the descriptor variables have been properly initialized.

Next let us consider two-dimensional arrays. Consider

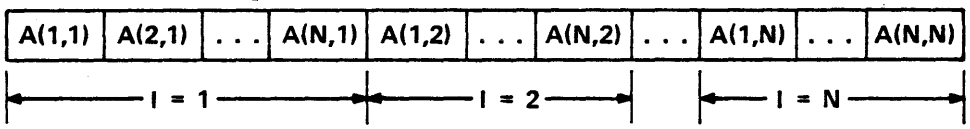
(2.11)
```
      DIMENSION A(N,N),B(N,N),C(N,N)
      .
      DO 9000 I = 1,N
      DO 9000 J = 1,N
C(J,I) = A(J,I) + B(J,I)
9000  CONTINUE
```

It is clear that the inner loop is a "vectorizable" kernel. Indeed, the code

(2.12)
```
      DO 9000 I = 1,N
    C(1,I;N) = A(1,I;N) + B(1,I;N)
9000  CONTINUE
```

eliminates one level of nesting and executes with mean vector length N.

However, a moment's reflection will reveal that for the purposes of the DO loop (2.11), the total data structure involved in the kernel is one vector for each array. Consider



Thus the entire DO loop (2.11) can be reduced to

(2.13)
```
C(1,1;N*N) = A(1,1;N*N) + B(1,1;N*N)
```

or even

(2.14)
```
CD = AD + BD
```

The vector version (2.13) executes with vector length N^2. This results in greatly improved vector performance. The reader is invited to check this fact, say for N = 100.

If all the data structures encountered in applications programs were contiguous, our discourse would be over. The straightforward descriptor syntax could be used to generate vector instructions and all problems would be solved!!

There are, of course, three main reasons why this scenario is naive.

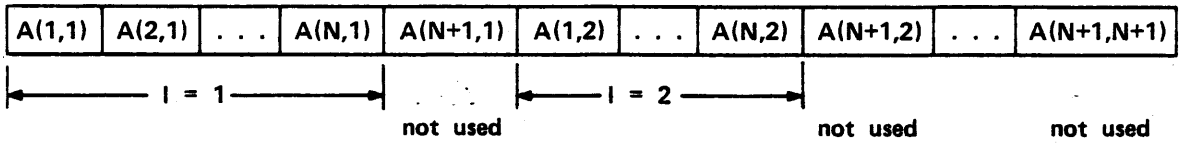
1. Data structures are often non-contiguous.
2. Arithmetic may depend on conditional statements.
3. Even when vectors exist, their existence may be masked by the scalar expression of an algorithm.

Points #1 and #2 are syntactic in nature and will be discussed in the remainder of this section. Point #3 is more semantic in nature and will be discussed to a limited extent in the next section.

Let us consider the same double DO loop (2.11) with the following declarative

(2.15)
```
DIMENSION A(N+1,N+1),B(N+1,N+1),C(N+1,N+1)
```

While the vectorized version (2.11) still is valid, it would seem that the superior solution (2.13) is no longer possible. This lowers the mean vector length from N^2 to N.



Since the majority of data elements of the arrays is active, it seems a shame to lose the more effective vector length. Indeed, since each of the three arrays has the same activity pattern

if only one could suppress the arithmetic on every (N+1)st element, the higher megaflop rate could be restored.

This brings us to the first vector primitive other than basic arithmetic.

(2.16) **Def. Control Store** – Given i. A source arithmetic vector pointed to by AD

$$AD \triangleright A_1, A_2, A_3, \ldots, A_N$$

Given ii. A source bit vector pointed to by BITD

$$BITD \triangleright b_1, b_2, b_3, \ldots, b_N$$

Given iii. A target arithmetic vector pointed to by BD

$$BD \triangleright B_1, B_2, B_3, \ldots, B_N$$

the process of control store stores Aj over Bj whenever bj is '1' and leaves Bj untouched when bj is '0', all at vector speed.

The control store operation can be done as a separate operation which executes at the same rate as vector addition. It is also possible to do the control store as part of a vector arithmetic instruction. When done in this fashion, it literally leaves the execution time invariant.

In the FORTRAN environment, the control store can be affected with the syntax

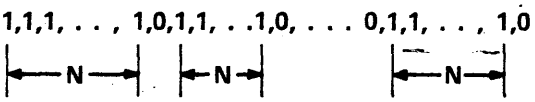
(2.17) **BD = Q8VCTRL(AD,BITD;BD)**

where AD is a descriptor for the source vector, BITD is a descriptor for the controlling bit vector and BD is a descriptor for the target vector.

When done as part of an arithmetic operation, at the present time, there is a "special call syntax" which allows the FORTRAN programmer to imbed META (CYBER 200 assembly language) in a FORTRAN program. In particular, the DO loop (2.11) with the effective declarative (2.15) can be reduced to one FORTRAN "special call syntax" statement, namely

(2.18) **CALL Q8ADDNV(,,AD,,BD,BITD,CD)**

where BITD is a descriptor pointing to a bit vector consisting of



and AD,BD,CD are descriptors pointing to the entire arrays as vectors.

Thus we are able to operate on the non-contiguous data structure with almost full efficiency. The "extra work" we must perform includes the bit vector construction and the redundant arithmetic.

As for the bit vector, it is possible to construct this particular one with one instruction which translates to one line of FORTRAN. This instruction is very efficient, and in any case, bit vectors of this type have values which are structure dependent rather than value dependent. Hence the values in the bit vector do not change over the course of execution.

As for the redundant arithmetic, intuitively it is clear that the half cycle spent doing the addition on each (N+1)st element whose storage to memory is suppressed by the controlling bit vector, is a small price to pay to avoid the extra vector startup.

Quantitatively, let

S = Startup time for a vector addition

R = Stream rate for a vector addition

A,B,C be dimensioned (N+M)x(N+M)

Then the control store technique is superior to the naive technique of restarting the vector whenever

$S/R > M$

Since S/R is between 150 and 200, if M is small, as in our little example, clearly the control store technique is superior.

The control store technique is also useful in vectorizing IF test controlled arithmetic done inside DO loops.

(2.19) **Def. Vector Relational** — Given i. Two source arithmetic vectors pointed to by AD and BD

$$AD \triangleright A_1, A_2, A_3, \ldots, A_N$$

$$BD \triangleright B_1, B_2, B_3, \ldots, B_N$$

Given ii. A target bit vector pointed to by BITD

$$BITD \triangleright b_1, b_2, b_3, \ldots, b_N$$

the statement

(2.20)
```
BITD = AD.GT.BD
```

will cause Aj to be compared with Bj at vector processing speed. If Aj is strictly greater than Bj, bj will be set to '1' and conversely.

Thus "decision structures" can be constructed at vector speed. Bit vectors so constructed can be used to control arithmetic.

Consider the following simplified one-dimensional flux trading model. It is assumed that an array of flux rates DXDT(J) J=1,N exists. If the model predicts a flux rate over a time step which is too large, one option is to pass no flux that time step. Thus

(2.21)
```
      DO 9000 J=1,N
    FLUX(J) = DT*DXDT(J)
      IF(FLUX(J).GT.X(J)) GO TO 9000
    X(J) = X(J) - FLUX(J)
9000  CONTINUE
```

NOTE THAT THE ORDER OF EVENTS IN SCALAR IS

ARITHMETIC	DECISION	· · ·	ARITHMETIC	DECISION	, ETC.

Thus the scalar code forces the computer to change its disposition with each operation from arithmetic processor to decision maker. The key to vector processing in this kernel is the ability to make the computer perform all of a given kind of operation "at once".

Using implicit descriptors, the vectorized form of this kernel is:

(2.22)
```
FLUXD = DT*DXDTD
BITD = FLUXD.LE.XD
CALL Q8SUBNV (,,XD,,FLUXD,BITD,XD)
```

The first line calculates all the fluxes with one vector operation. The second line builds the decision structure, i.e., the bits pointed to by BITD contain the outcomes of all the IF tests. This is again one vector operation. The third line trades the fluxes whenever possible with one vector operation.

As for efficiency, one assumes in a well behaved model that the percentage of 1's in the bit vector is high.

Let me make two final comments on this kernel. First, if one suspects that the model is behaving poorly, i.e., the bit vector is mostly 0's, one can ascertain very easily how many 1's there are. There are several efficient alternatives. We will not speak of them here. Secondly, the scalar DO loop (2.21) compiles into 12003 instructions. The vector kernel compiles into 3 instructions.

While the control store is an invaluable tool, the following kernel illustrates the need for what we shall call "data motion" primitives.

Suppose it is desired to evaluate a polynomial

(2.23)

$$Y = \sum_{k=0}^{\ell} A_{\ell-k+1} X^k$$

for various values of X which are evenly spaced within an array. Using Horner decomposition, such a kernel might be written

(2.24)

```
      DO 9000 J=1,N,M
Y(J) = A(1) * X(J)
      DO 9010 K=2,L
Y(J) = Y(J) + A(K)
Y(J) = X(J) * Y(J)
9010  CONTINUE
Y(J) = Y(J) + A(L+1)
9000  CONTINUE
```

If M=1, this is clearly a vectorizable kernel:

(2.25)

```
YD = A(1)*XD
      DO 9010 K=2,L
YD = YD+A(K)
YD = XD*YD
9010  CONTINUE
YD = YD+A(L+1)
```

If M>1, one can write the above kernel using special call syntax with the appropriate bit vector and control store the arithmetic.

Note that, independent of M, 2LN arithmetic operations will be done. Thus, for M=2, the peak result rate will be half the peak hardware rate.

The problem here is that the data structure is "fractured". Indeed only 1 out of each M elements is active.

Perhaps it would be better to get the active elements together. For this particular kernel, we should investigate the compress.

(2.26) **Def. Compress** — **Given i.** **A source data vector pointed to by AD, ie.**

$$AD \triangleright A_1, A_2, A_3, \ldots, A_N$$

And a bit vector pointed to by BITD, ie.

$$BITD \triangleright b_1, b_2, b_3, \ldots, b_N$$

Given ii. **A target data vector pointed to by BD, ie.**

$$BD \triangleright B_1, \ldots, B_l$$

The statement

BD = Q8VCMPRS (AD,BITD; BD)

will cause those elements of the vector that correspond to 1's in the bit vector to be compressed into the B vector.

Concomitantly there is an operation called expand which allows one to put the answers back into the proper places in the array.

The alternative to the control store algorithm is to compress the X values into a temporary vector, do full efficiency vector arithmetic and expand the answers. In comparing the two approaches we shall assume that N is large enough to ignore startup time. We also note that the pair

```
YD = YD + A(K)

YD = YD * XD
```

is a linked triad and can be calculated at one cycle per result per pipe on the CYBER 203E.

The following chart contains the cycle count (20 nsec. cycles) for the stream time for the two approaches on various architectures.

(2.27) RECALL N = LENGTH OF ARRAY
M = DISTANCE BETWEEN ACTIVE ELEMENTS
L = DEGREE OF POLYNOMIAL

	CYBER 203	2 PIPE CYBER 203E	4 PIPE CYBER 203E
CONTROL STORE	3NL	$\frac{N}{2}(L+1)$	$\frac{N}{4}(L+1)$
COMPRESS	$9N + \frac{3LN}{M}$	$N + \frac{N(L+1)}{2M}$	$\frac{N}{2} + \frac{N(L+1)}{4M}$

For M >1, as L gets larger it is clear that the compress gets more efficient. This is true since the compress-expand time can be amortized over more arithmetic operations, whereas the redundant calculation in the control store technique varies directly with the total arithmetic operation count. In particular, one can construct the following table of crossover values, i.e., those values of L at which the compress-expand technique becomes superior for fixed values of M.

(2.28)

	STAR 100	2 PIPE CYBER 203E	4 PIPE CYBER 203E
M = 2	L = 6	L = 3	L = 3
M = 5	L = 4	L = 2	L = 2
M = 10	L = 3	L = 2	L = 2

Finally, while the exact algorithm we have analyzed is not that common, it is illustrative of a large class of syntactic kernels, i.e., those that require much intermediate arithmetic on non-contiguous data. Neither the polynomial form nor the periodicity of data were essential to the discussion.

Another data motion primitive is the merge.

(2.29) **Def. Merge –** **Given i.** **Two source data vectors pointed to by AD and BD. And a bit vector pointed to by BITD.**

Given ii. **A target vector pointed to by CD, the statement**

CD = Q8VMERG(AD,BD,BITD; CD)

will examine consecutive bits in the bit vector. If the bit is '1', the next unused element of the A vector will be placed in the next contiguous element in the C vector. Concomitantly, if the bit is '0', the next unused element of the B vector will be placed in the C vector.

For example, if

$AD \triangleright A_1, A_2, A_3, A_4$

$BD \triangleright B_1, B_2, B_3, B_4, B_5$

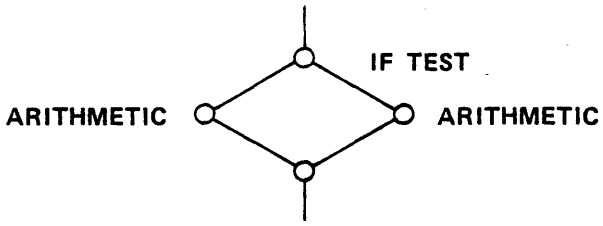
$BITD \triangleright 1\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 0$

Then after merging,

$CD \triangleright A_1, B_1, B_2, A_2, A_3, A_4, B_3, B_4, B_5$

This operation allows one to vectorize kernels with IF tests by both the control store and compress-merge approaches.

Consider the following calculation tree



Where the arithmetic depends on the results of an IF test, the whole tree being inside a DO loop.

The IF test translates to a vector relational creating a bit vector. Then one can do all the arithmetic in both branches using the bit vector to control store results; or one can use the bit vector to compress out active data, do full efficiency vector arithmetic on the active data, and conclude with a merge back into the original data structure.

The relative efficiency of one approach over the other depends on such factors as vector length and the amount of arithmetic in each branch.

There are two limitations to the control store and compress-merge techniques

1. The output data stream elements are in the same order as the input data, i.e., if Aj occurs before A_k in a data structure before compressing or merging, the same will be true after the given operation.

2. The time to process a control store, compress or merge is proportional to the total data structure size. For very sparse active data this exacts a high price.

Let us illustrate the second point with an example that more properly belongs to the next section.

Let A be an N X N matrix stored by columns, i.e., standard FORTRAN storage. Suppose we wish to operate on a row of the matrix, say to equilibrate it. One could compress out every Nth element from the N^2 elements. This would take 0 (N^2) cycles even though we seek only N elements.

To overcome these limitations, there is the capability of scatter/gather.

(2.30) **Def. Gather — Given:** **A source data vector pointed to by AD, an index list, ie., an integer data vector pointed to by ID and a target data vector pointed to by BD, the statement**

BD = Q8VGATHR (AD, ID; BD)

will use the integers pointed to by ID as indices to take possibly disparate elements from the A vector and make them contiguous in the B vector.

Essentially, it accomplishes the following:

(2.31)

```
      DO 9000 J=1,N
    B(J) = A(I(J))
9000  CONTINUE
```

and does it in 0 (N) cycles.

Concomitantly, there is Q8VSCATR to accomplish

(2.32) **A(I(J)) = B(J)**

The constant of proportionality in the O(N) time to gather or scatter changes radically with succeeding architectures

(2.33)

ARCHITECTURE	TIME PER FETCH	RATIO
STAR 100 MICRO-CODED INSTRUCTION	800 ns	1
CYBER 203 SCALAR L/S ALGORITHM	100 ns	8
CYBER 203E MICRO-CODED INSTRUCTION	30 ns	26

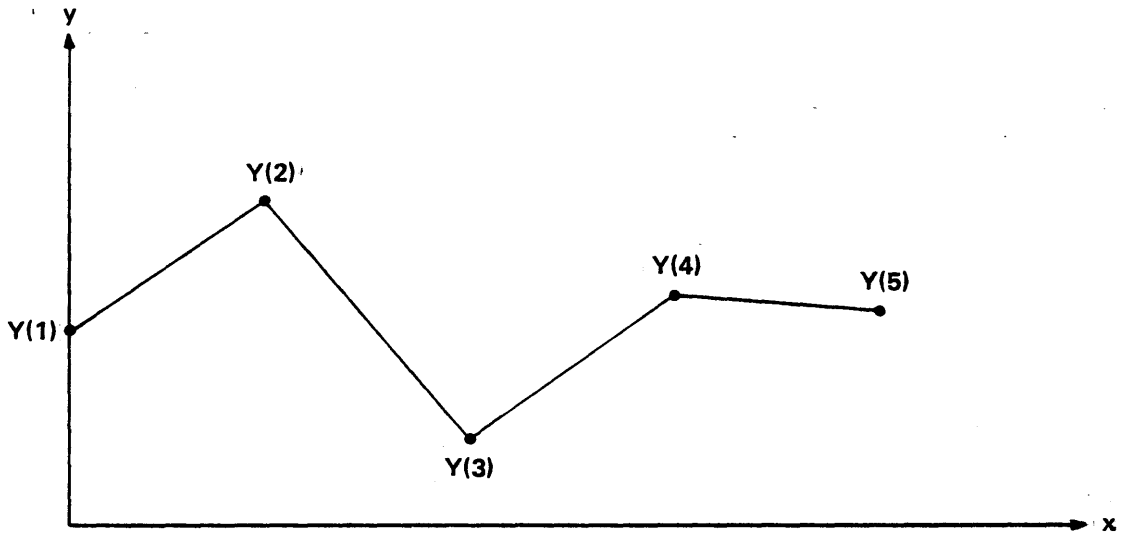
This tremendous efficiency increase will be needed as we move to three-dimensional data structures, gathering and scattering two-dimensional and one-dimensional substructures.

For our final kernel in this section, let us consider a subroutine to calculate a piecewise linear function on a vector of data.

Let $f(n\Delta) = Y(n+1)$ $n = 0,N$ and for $n\Delta < X < (n+1)\Delta$

$$f(X) = Y(n+1) + (X-n\Delta)\ \frac{(Y(n+2) - Y(n+1))}{\Delta}$$

We wish to calculate $Z(J) = f(X(J))$ $J = 1,N$ where the values of $X(J)$ are not in any pre-assigned order. Thus, we must gather the ordinate values "to the left and right".



If we multiply the X values by $1/\Delta$, the resulting vector elements will have an integral part which is equal to $n\Delta$ such that $n\Delta < X < (n+1)\Delta$ and hence the fractional part is $(X-n\Delta)$.

Propitiously, the operation "greatest integer less than or equal to" is one vector instruction on the CYBER 200. Hence, the algorithm can be completely vectorized.

1. Multiply X vector by $1/\Delta$
2. Find "floor" of resulting vector and use as index list for gather
3. Subtract to get vector of $(X-n\Delta)$'s

4. Gather left hand ordinate values
   Gather right hand ordinate values

5. Evaluate f(X) with simple vector arithmetic

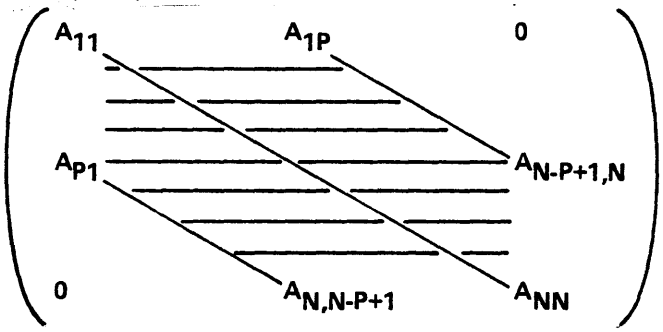
Further syntactic details of data motion can be found in (3) and (5).

3. VECTOR NUMERICAL LINEAR ALGEBRA

Matrices can be divided into four classes depending on the structure of sparsity, i.e., the number and placement of zeros.

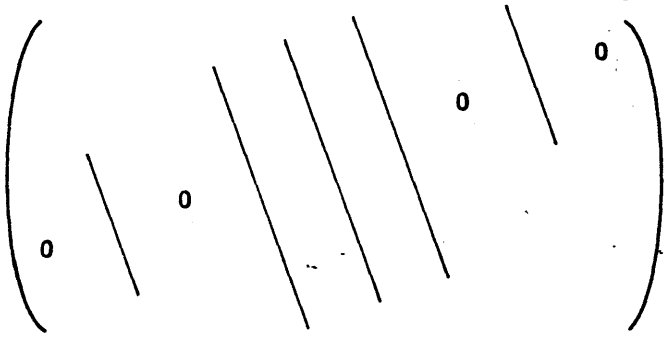
1. Full matrices - Those matrices where one may not assume any given element is a priori zero

2. Large banded - Those matrices that are N x N such that there exists $P << N$ with $A_{k,j} = 0$ for $|k-j| \geq P$. Pictorially such matrices look like



where the hatched-in region contains the non-zeros. Usually $P \rightarrow \infty$ as $N \rightarrow \infty$.

3. Structured sparse - Those matrices that are N x N with 0(N) non-zeros placed in a structured pattern, usually along a fixed bounded number of diagonals. Pictorially such matrices look like



4. Random sparse - Those matrices that are N x N with 0(N) non-zeros placed in a random pattern.

The various sparsity regimes just outlined demand widely differing data structures for optimal execution of various numerical linear algebraic algorithms on a vector processor. While this fact is true and is being recognized within the computing community, there is also a belief that this is not true for scalar processors. Before we get on to the proper discussion of this section, let me address this topic.

It is not true that one can ignore data structure on a scalar processor and get optimal performance. There are basically two reasons why this misbelief is widely held.

1. The difference in performance on a scalar processor between "casual" FORTRAN and optimal FORTRAN is usually much less than an order of magnitude. Thus the most one can procure for one's efforts is usually a factor of 2 or 3.

2. The indexing nomenclature of FORTRAN would lead one to believe that accessing a multi-dimensional data structure is painless regardless of the order of acquisition.

The fact of the matter is that one can improve scalar performance by carefully indexing within a data structure. But the improvement to be attained is not nearly so pronounced as on a vector processor. But rather than saying that a scalar processor can execute the primitive operations (such as loading, arithmetic, storing, etc.) as well for structured sparse problems as for full problems, it is true that to a certain extent it executes the primitive operations for a full problem as poorly as it does for a structured sparse problem.

The final truth is that for a contiguous data structure, the processor can maximize its ability to exchange data with memory and process it. The vector hardware of the CYBER 200 is designed around this basic truth.

We shall discuss various fundamental algorithms for matrices in sparsity regimes #1-3. Even on scalar processors, regime #4, the random sparse regime, needs very careful attention. As of yet it is not possible to vectorize operations in this regime to the same extent as the other three.

Let us begin with matrix addition and multiplication for full matrices.

If two full matrices are stored in the same order as contiguous data structures, we have already seen that the matrix sum, being an element by element sum can be accomplished with one instruction provided the size is 255x255 or less. However, even for a 1000x1000 case one would need only 16 instructions. In particular, for columnar or row storage, (2.13) and (2.14) illustrate this fact.

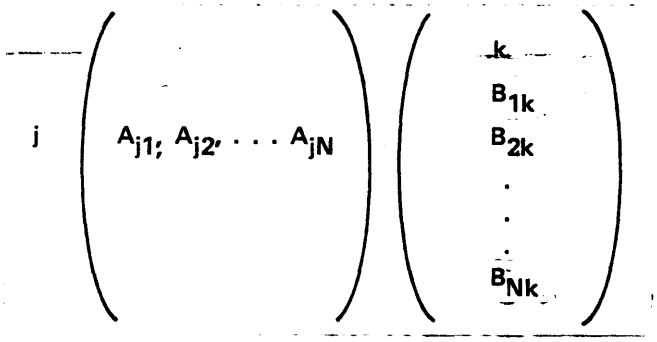
Matrix multiplication is not quite so trivial a matter. Let us recall the definition of the product of two NxN matrices:

Given full matrices A,B both N x N then for C = AB,

$$C_{jk} = \sum_{l=1}^{N} A_{jl} B_{lk}$$

If one examines (3.1) one can see that the j,k element of C is the inner (or DOT) product of the jth row of A and the kth column of B. This naturally leads to our first algorithm. If A is stored by rows and B by columns, the product matrix can be formed in any storage fashion with N^2 inner products.

(3.1)



On the CYBER 203, the timing for the inner product instruction is

12N + 260 (20 nsec.) cycles

while for the CYBER 203E it is projected to be

N + 0(1) cycles.

Thus, the total time for the first algorithm is

CYCLE COUNT

CYBER 203 $\quad \sim N^3 + 0(N^2)$ CYBER 203

2 PIPE CYBER 203E $\quad \sim N^3 + 0(N^2)$ CYBER 203E

4 PIPE CYBER 203E

If A is not stored by rows or B by columns, one must gather the appropriate entities. This will affect only the $0(N^2)$ term of course. However, one should not expect to find this particular storage order too often. One would expect that both would be stored one way or the other.

Is it possible to find an algorithm that dispenses with the necessity of the gathers and yet is efficient? Indeed there is. Not only is it possible to do away with the gathers, but asymptotically this second algorithm we shall consider will be four times faster than the first algorithm!!

It is based on the linear algebraic fact that

(3.2)

$$\begin{pmatrix} C_{1K} \\ \cdot \\ \cdot \\ \cdot \\ C_{NK} \end{pmatrix} = B_{1k} \begin{pmatrix} A_{11} \\ \cdot \\ \cdot \\ \cdot \\ A_{N1} \end{pmatrix} + B_{2k} \begin{pmatrix} A_{12} \\ \cdot \\ \cdot \\ \cdot \\ A_{N2} \end{pmatrix} + \cdots + B_{Nk} \begin{pmatrix} A_{1N} \\ \cdot \\ \cdot \\ \cdot \\ A_{NN} \end{pmatrix}$$

This leads to the so called "outer product" algorithm, namely:

If A is stored by columns, B is stored in any fashion then

(3.3)

```
            DO 9000  K = 1,N
      C(1,K;N) = B(1,K) * A(1,1;N)
            DO 9010  L = 2,N
      C(1,K;N) = C(1;K;N) + B(L,K) * A(1,L;N)
9010        CONTINUE
9000        CONTINUE
```

will compute C and store it by columns.

Note that the vector statement in the inner loop is a linked triad.

MEAN VECTOR LENGTH

The time for this algorithm is

$3N^3 + 0(N^2)$	CYBER 203
$N^3/2 + 0(N^2)$	CYBER 203E two pipe
$N^3/4 + 0(N^2)$	CYBER 203E four pipe

Not only is this algorithm asymptotically four times faster than the first algorithm, but the $0(N^2)$ is much smaller due to the fact that no data motion is required.

The reader versed in linear algebra will immediately see that the dual form of (3.2) which expresses the Kth row of C as a linear combination of the rows of B yields an algorithm identical in performance to (3.3). This algorithm demands that B be stored by rows and calculates C by rows, but places no restrictions on A.

To sum up, the following table presents the necessary data structures to achieve the matrix operations listed with the mean vector length given on various architectures.

(3.4)

	FULL MATRIX ADDITION C = A + B	FULL MATRIX MULTIPLICATION C = AB
DATA STRUCTURE	A rows A cols. B rows B cols. C rows C cols.	A cols. A * B * B rows C cols. C rows
	N^2	N
	$N^2 + 0(1)$ $N^2/2 + 0(1)$ $N^2/4 + 0(1)$	$3N^3 + 0(N^2)$ $N^3/2 + 0(N^2)$ $N^3/4 + 0(N^2)$

Since the mean vector length for the multiplication algorithms is the length of a column of the matrix, it is clear that the data for the entire matrix need not be stored contiguously, but merely the data for the columns (or rows). In particular, if an augmented matrix is stored by rows, this will not affect the efficacy of the algorithm.

Next let us discuss the operations of matrix addition and multiplication for the other two sparsity regimes.

We shall start with the structured sparse, since it will allow us to introduce a new storage order.

For a structured sparse matrix, and in certain circumstances for a large banded matrix, the preferred storage order for matrix multiplication is diagonal storage. The number of non-zeros in each row or column is small, usually less than ten. Hence there is no way to get efficient vectorization with this sparsity. However, the number of elements in the non-zero diagonals varies from N to N-P+1. This is indeed more like what we seek. Can this storage order give rise to an efficient algorithm, though? The answer is yes. The details are left to the reader who may consult (7). We shall illustrate this fact with the product of two tridiagonal matrices.

These matrices are such that

$$A_{K,J} = 0 \text{ unless } |K-J| \leq 1.$$

The product of two tridiagonal matrices is pentadiagonal. The main diagonal terms of the product are given by

$$C_{KK} = A_{K,K-1}\, B_{K-1,K} + A_{KK}\, B_{KK} + A_{K,K+1}\, B_{K+1,K} \quad K=1,N \tag{3.5}$$

$$\text{where } A_{10} = B_{01} = A_{N,N+1} = B_{N+1,N} = 0.$$

as K runs from 1 to N

C_{KK}	traverses	Main diagonal of C
$A_{K,K-1}$	traverses	Sub diagonal of A
$B_{K-1,K}$	traverses	Super diagonal of B
A_{KK}	traverses	Main diagonal of A
B_{KK}	traverses	Main diagonal of B
$A_{K,K+1}$	traverses	Super diagonal of A
$B_{K+1,K}$	traverses	Sub diagonal of B

Hence (3.5) directly translates into vectorized CYBER 200 FORTRAN. The exact form will depend upon how the descriptors for the various diagonals are set up.

Large banded matrices lie somewhere between the extremes of full and structured sparse. We shall see that both of the storage orders we have discussed (row-column, diagonal) can be efficiently applied to this sparsity regime. If all we were interested in were addition and multiplication of matrices, the diagonal order would be preferred. However, usually one is interested in the solution of the linear system embodying the matrix. For this, row-column storage is preferred. Again, if the large banded matrix is formed from a product of, say tridiagonal matrices, the diagonal order is preferred. The individual choice must be made on the basis of more exact problem details.

As for the storage orders themselves, it should be clear that the algorithm illustrated for the product of tridiagonal matrices goes over to large banded matrices. Indeed, diagonal storage can even be used for full matrices. The stream time for multiplication in this case will be the same as it was for the row-column algorithm, but the startup time will be longer. One can see this qualitatively by noting that each of the vectors in the row-column algorithm have length N while in the diagonal algorithm lengths will vary from one to N.

The row-column algorithm for full matrices depended upon having contiguous non-zeros in a given row or column. In the full case, there are N of them. In the large banded case, there are between P and 2P-1. Hence the row-column algorithm goes right over with a mean vector length of $2P-0(P^2/N)$ which we assume is $2P-0(1)$. There is some ancillary bookkeeping to do to keep track of the top and bottom of the non-zeros, but no more so than for a scalar process.

As for matrix addition, the fundamental fact still holds, namely, if the two matrices have their non-zeros stored contiguously in the same order, whether row, column or diagonal, the sum can be calculated with one instruction.

Let us turn to the subject of linear equation solution. There are two basic approaches to this problem, the direct and the iterative. For full (and large banded) systems, we shall illustrate Gaussian Elimination and SOR, the most popular direct and iterative procedures. We shall see that each of these algorithms can be implemented on the CYBER 200 with great efficiency.

For structured sparse systems, the implementation demands a bit more subtlety. We shall illustrate Red-Black SOR, a workable iterative scheme for certain structured sparse systems and cyclic reduction, a direct scheme for point and block tridiagonal systems.

As with many direct methods, Gaussian Elimination is composed of two parts, the decomposition phase and the substitution phase. The decomposition phase consists of applying a succession of elementary transformations to the augmented matrix which result in upper semi-triangular form. The substitution phase consists in the solution of the derived upper semi-triangular system by uncoupling the variables one by one. Interchanging rows in the augmented matrix is an allowable elementary transformation and can be used to pivot.

Let us dispense with the question of pivoting once and for all. We shall be concerned with implementation of Gaussian Elimination for both a rowlike and a columnar storage scheme for the matrix.

Partial pivoting consists of two phases, the search and the row interchange. For rowlike storage, the search for the maximal element is a simple vector operation. Indeed the CYBER 200 has one instruction which searches for the maximum absolute value contained in a vector, returning the value and its location. The row interchange is accomplished with two gathers and two scatters. For rowlike storage, a gather is necessary before the search can be done, but the row interchange is a succession of simple vector moves.

The decomposition consists of two parts, the multiplier formation and the reduction. These phases are repeated (N-1) times until the matrix is transformed into upper semi-triangular form. Le us illustrate the first occurrence of these phases.

Consider the matrix in original form:

(3.6)
$$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1N} \\ A_{21} & & & \\ \vdots & & & \\ A_{N1} & & & \end{pmatrix}$$

We wish to find a set of multipliers such that when the first row is multiplied by the Kth multiplier and added to the (K+1)st row, the (K+1)st element in the first column will become zero. Provided that A_{11} is not zero, it is clear that

(3.7) $$- A_{21}/A_{11}, - A_{31}/A_{11}, \ldots, - A_{N1}/A_{11}.$$

is such a set of multipliers. The problem of A_{11} being zero or even very small is part of the problem of pivoting which we have already considered.

Thus, for the multiplier formation, the columnar storage is favored since (3.7) can be formed with one vector multiplication without the need for a gather.

Before we discuss the reduction phase, let us distinguish two types of rowlike storage for the augmented matrix. Let M be the number of right hand sides to be solved for. The most common values for M are one and N. When M = N we are usually solving for the inverse of the matrix A. When M = N we will consider two cases of rowlike storage. In the standard case, the matrix A and the matrix of right hand sides will be considered to be separate. In the augmented case, the augmented matrix A:Y will be stored by rows. Thus the following elements will be contiguous:

$$A_{K,1}\ A_{K,2} \cdots A_{K,N}\ Y_{K,1}\ Y_{K,2} \cdots Y_{K,M}$$

The first reduction phase entails multiplying the first row by the succession of multipliers and adding the results to the succession of rows below.

This has traditionally been presented as a rowlike algorithm, i.e., one takes one multiplier, say $(-A_{21}/A_{11})$ and multiplies the whole row $A_{12} \cdots A_{1N}$ and adds to the whole row $A_{22} \cdots A_{2N}$.

Indeed, this operation is a linked triad on the CYBER 203E, and if we have augmented rowlike storage, the entire row of A and Y can be treated with one vector instruction.

However, when one investigates the total work to be done in treating the minor of the augmented matrix, there is no reason why one cannot use all the multipliers, multiply by one element in the first row and add this vector to the respective column in A. Similarly for Y_{1K}.

(3.8)

$$\begin{pmatrix} \frac{-A_{21}}{A_{11}} \\ \cdot \\ \cdot \\ \cdot \\ \frac{-A_{N1}}{A_{11}} \end{pmatrix} * A_{1K} + \begin{pmatrix} A_{2K} \\ \cdot \\ \cdot \\ \cdot \\ A_{NK} \end{pmatrix}$$

Thus, the reduction phase is just as much a columnar algorithm. Perhaps the reason why this a posteriori obvious fact has not been more generally recognized is that traditionally numerical analysis texts have considered matrices as rowlike objects. (It is curious that given this fact, FORTRAN standards treat matrices as columnar objects when mapping them onto essentially one-dimensional memories.)

Since the decomposition phase has an $0(N^3)$ operation count, while the substitution phase has an $0(N^2)$ count, if M, the number of right hand sides, is small, then decomposition will clearly be the dominant phase. If, however, M = N, then substitution also becomes an $0(N^3)$ phase. Thus we will also consider the vectorization of substitution.

First, let us summarize the vectorizability of decomposition:

(3.9)

	Column Storage	Row Storage	Augmented Row Storage
Multiplier Formation	Vectors — Mean Length N/2	Gather Necessary	Gather Necessary
Reduction M = 1	Vectors — Mean Length 2N/3	Vectors — Mean Length 2N/3	Vectors — Mean Length 2N/3 + 1
Reduction M = N	Vectors — Mean Length 2N/3	Vectors — Mean Length 5N/6	Vectors — Mean Length 5N/3

Thus for M = 1 columnar storage is clearly superior, while for M = N, the extra time to gather is more than offset by the longer mean vector length during reduction. Of course, when M = N, we must consider the substitution phase before deciding since in this case the two phases have comparable operation counts.

At the conclusion of the decomposition, the matrix is in upper semi-triangular form. If the multipliers have been saved, then the LU decomposition has been affected.

The substitution phase is recursive in that one solves for

$$X_{NK} = \frac{Y_{NK}}{A_{NN}} \qquad K=1,M$$

and then

$$X_{N-1,K} = \frac{1}{A_{N-1,N-1}} \left(Y_{N-1,K} - A_{N-1,N} X_{N,K}\right) \quad K=1,M \text{ etc.}$$

This presents no problem for rowlike storage for M large. Indeed, the two equations above represent simple vector operations of length M. The problem of M=1 or column storage seems to leave us with an inner product algorithm, i.e.,

$$X_{N-L} = \frac{1}{A_{N-L,N-L}} \left(Y_{N-L} - A_{N-L,N} X_N - A_{N-L,N-1} X_{N-1} \ldots\right)$$

However, for column storage, if we observe carefully, it is possible to perform the substitution phase using vector linked triads.

The key insight is that when X_N is solved for, its affect on all the variables $X_1, \ldots, X_{N-1}$ can be "put back" into the equations. This can best be seen by looking at the calculational tableau for a small case, say N=4.

$$X_4 = \frac{1}{A_{44}} \left(Y_4\right)$$

$$X_3 = \frac{1}{A_{33}} \left(Y_3 - A_{34} X_4\right)$$

$$X_2 = \frac{1}{A_{22}} \left(Y_2 - A_{24} X_4 - A_{23} X_3\right)$$

$$X_1 = \frac{1}{A_{11}} \left(Y_1 - A_{14} X_4 - A_{13} X_3 - A_{12} X_2\right)$$

After solving for X_4, one can multiply the column A_{14}, A_{24}, A_{34} by X_4 and subtract from the column Y_1, Y_2, Y_3. This is a linked triad and when completed, the tableau is

$$X_3 = \frac{1}{A_{33}} \quad (Y_3)$$

$$X_2 = \frac{1}{A_{22}} \quad (Y_2 - A_{23} X_3)$$

$$X_1 = \frac{1}{A_{11}} \quad (Y_1 - A_{13} X_3 - A_{12} X_2)$$

The above process can be repeated until all X's are solved for.

Let us then summarize the vectorizability of substitution:

(3.10)

	Column Storage	Row Storage
M = 1	Vectors — Mean Length N/2	Inner Product Necessary
M = N	Vectors — Mean Length N/2	Vectors — Mean Length N

This only confirms what we discovered about decomposition, namely for $M = 1$ columnar storage is clearly superior, whereas for $M = N$ the situation is reversed, especially if augmented row storage is possible.

We have been discussing alternative storage schemes to optimize the vectorization of Gaussian Elimination. But is it possible to make a choice in general? The answer to this question depends upon analysis of the whole program. In particular, if the matrix A is formed by scalar code, one can arrange its storage order as one pleases. Even in certain vectorized cases one can still choose. Such considerations are global in nature rather than local. To get the best performance, one cannot look at one DO loop or even one subroutine, but at the usage of the data structure as a whole.

If we consider the foregoing discussion for the case of large banded systems, it is clear that for both storage orders the preceding analysis still holds, but with vector length P replacing N/2 and 2N/3 in the various phases. The efficiency in this case is determined by the size of P.

It is also clear the foregoing analysis cannot efficiently be applied to structured sparse systems. We shall return to this question a little later.

Let us go back to a full system which we wish to solve iteratively. There are many methods which have been evolved to attack this problem. But there are several basic considerations which are common to most or all of these methods. We shall illustrate these basics, starting with the Jacobi and SOR algorithms for a full system.

Since SOR differs from Gauss Seidel only in the addition of a multiple of the diagonal to the right hand side, without loss of generality we shall consider the Jacobi and Gauss Seidel algorithms.

The basis for an iterative solution to $AX = Y$ is to decompose $A = M-N$ and attempt to find X such that $MX = NX+Y$.

This "fixed point" is sought by making a guess $X^{(0)}$ and repeating the algorithm $MX^{(n+1)} = NX^{(n)} + Y$ until $X^{(n+1)}$ and $X^{(n)}$ are very close in the appropriate norm. Note that the iterative technique trades solving $AX=Y$ once for solving $MX^{(n+1)} = NX^{(n)} + Y$ many times.

Much of the variety of iterative methods comes from the various ways one can form M,N such that $M - N = A$. At the heart of many of these is the following additive decomposition of A

$$A = A_D + A_L + A_U$$

Where AD is the diagonal portion of A, A_L is the lower triangular portion and A_U is the upper triangular portion.

In terms of the notation above, the Jacobi iterative scheme is characterized by

$$\text{Jacobi} \quad M = A_D \quad N = -(A_L + A_U)$$

This method typifies those methods that do not update variables until the end of a given iteration. It is also well known that for a large class of problems it is the slowest converging method in terms of the number of iterations necessary for sufficient accuracy. It is not our business here to discuss this. Our object is to discuss the vectorization of the scheme.

For full A, Jacobi vectorizes with length N. All one has to do is have a copy of the matrix A with the main diagonal zeroed out. Then the multiplication of $-(A_L + A_U)$ times $X^{(n)}$ is multiplication of a full matrix times a vector which we already know can be vectorized with a linked triad algorithm with vector length N. One simply adds Y and the right hand side is formed.

The solution of $A_D X^{(n+1)} = RHS$ is a trivial vector multiply by the vector of reciprocals of diagonal elements of A.

By contrast the Gauss Seidel iterative scheme is characterized by

Gauss Seidel $M = A_D + A_L \quad N = -A_U$

This method typifies those that do update variables as soon as a new value is available.

As for the vectorization, evaluation of the right hand side,

$$-A_U X^{(n)} + Y$$

Vectorizes to a linked triad algorithm with mean vector length N/2.

The solution of

$$(A_D + A_L) X^{(n+1)} = RHS$$

is recursive, but the form of the recursion is the same as the substitution phase of Gaussian Elimination. Hence it vectorizes to a linked triad algorithm with mean vector length N/2.

As before, these same algorithms work efficiently for large banded systems with suitable modifications and large enough P.

As for structured sparse systems, the Jacobi algorithm easily transports as long as $-(A_L + A_U)$ is stored by diagonals. However, the Gauss Seidel algorithm does not go over.

In summary, for the algorithms we have just presented

(3.11)

	Jacobi or JOR	Gauss Seidel or SOR
Full System	Vectors — Mean Length N	Vectors — Mean Length N/2
Large Banded System	Vectors — Mean Length 2P	Vectors — Mean Length P
Structured Sparse System	Vectors — Mean Length > $N-P_0$	Not Vectorizable

where P_0 is the non-zero diagonal furthest from the main diagonal.

It is fair to say that we have rather definitively treated the vectorization of the linear equation solution techniques just discussed, at least when the problem is main memory contained, for full and large banded systems. It is also fair to point out that except for the Jacobi iterative algorithm we have not given any results for structured sparse problems.

In a certain sense, there are no definitive answers here. That is not to say that there have not been algorithms developed, but that they are special case developments, geared to specific problems. Thus this is a very active area of research.

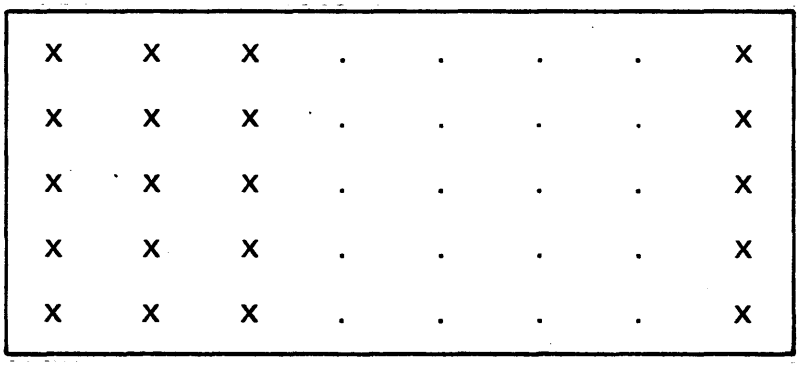
We will close with two specific problems, one to illustrate the vectorization of an interative method for a structured sparse problem and the other to illustrate a direct method.

In general, one does not use SOR type algorithms for full systems of equations. The typical usage of this scheme is for structured sparse problems arising in the solution of boundary value problems of partial differential equations.

Suppose one wishes to solve Poisson's equation

$$\nabla^2 \phi = \rho$$

with Dirichlet boundary conditions on a rectangle. Using a uniform mesh, one can consider the rectangular region as a sequence of nodes. Replacing ∇^2 by the sum of second order central differences in the X and Y directions, the partial differential equation is replaced by a system of linear equations:



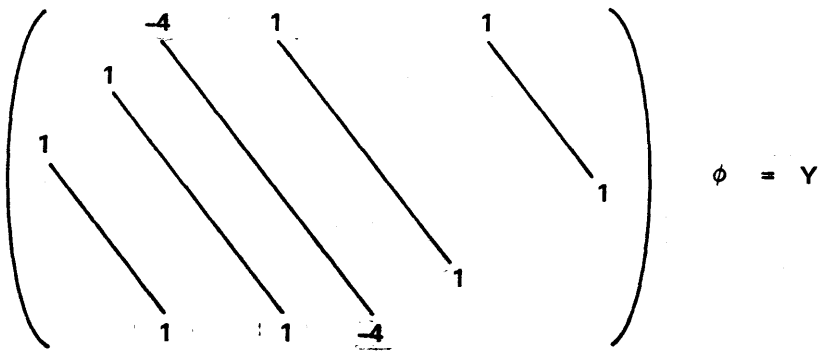
(3.12)
$$\frac{\phi_{j-1,k} - 2\phi_{j,k} + \phi_{j+1,k}}{\Delta^2} + \frac{\phi_{j,k+1} - 2\phi_{j,k} + \phi_{j,k-1}}{\Delta^2} = \rho_{jk}$$

Numbering the nodes in lexicographic order

1,1 2,1 N,1 1,2 ... N,N ,

the system of linear equations has the form

(3.13)



where Y is formed from the boundary values of ϕ and the values of ρ. This is an archtypal structured sparse system.

There are five non-zero diagonals, with three of them clustered around the main diagonal and the other two spaced apart from these by a distance equal to the number of nodes in one direction on the grid. This sparsity pattern defies vectorization with the algorithms introduced so far.

The key fact needed to vectorize this problem is that the matrix above is a consistently ordered 2-cyclic matrix.

One might know the 2-cyclic nature of this matrix from the fact that it is block tridiagonal, but this fact is geometrically recognizable if we go back to the mesh and ask ourselves what the SOR method implies.

Each iteration, one passes over all the nodes, updating the value for ϕ from those values "around it". If we order the nodes lexicographically, each node is updated from two values at the last iteration and two values of the given iteration. The recursion is obvious. What if we reorder the nodes, dividing them into two classes, say the red and the black.

(3.14)

X	0	X	.	.	.	.	.	.	X
0	X	0	.	.	.	.	.	.	0
X	0	X	.	.	.	.	.	.	X
0	X	0	.	.	.	.	.	.	0
X	0	X	.	.	.	.	.	.	X

It is geometrically clear that the red nodes are updated from the black nodes and conversely. Thus, with this ordering, it should be possible to vectorize the SOR.

If we go back to the algebraic problem and reorder the nodes so that all the black nodes precede the red ones, the system assumes the form

(3.15)

$$\left(\begin{array}{c|c} D_1 & B \\ \hline C & D_2 \end{array}\right) \left(\begin{array}{c} \phi_1 \\ \phi_2 \end{array}\right) = \left(\begin{array}{c} Y_1 \\ Y_2 \end{array}\right)$$

where D_1 and D_2 are diagonal matrices, B and C are structured sparse matrices, ϕ_1 and ϕ_2 are the ϕ values on the black and red nodes, respectively.

The Gauss Seidel algorithm applied to this reordered system yields:

(3.16)

$$D_1 \quad \phi_1^{(n+1)} = Y_1 - B \quad \phi_2^{(n)}$$

$$D_2 \quad \phi_2^{(n+1)} = Y_2 - C \quad \phi_1^{(n+1)}$$

Note that the structured sparse matrices are now on the righthand side, not the lefthand side. Thus, we have traded equation solution with a structured sparse system for matrix multiplication by a structured sparse matrix. This latter operation is eminently vectorizable if B and C are stored by diagonals.

For a general discussion of SOR methods see (9) and (10). For implementation information on a red-black problem see (8).

Finally, we must address the question of convergence. When we rearrange the order of the variables in a scheme such as SOR which updates as soon as possible, there is at least the possibility that the convergence rate may be altered, if not destroyed.

However, in this case it is known that both the lexicographic order and the red-black order are consistent orders, and thus the spectral radius of the respective SOR iterative matrices are equal which implies that the asymptotic convergence rates are equal. Of course this does not guarantee that one will see identical convergence rates for a finite number of iterations. Experiments in a large number of cases has demonstrated heuristically that actual rate of convergence for the red-black ordering is no worse than the lexicographic. Indeed, in most cases it has been slightly better from the standpoint of number of iterations. Of course the speedup in time per iteration is phenomenal.

Finally, it should be pointed out that the same red-black ordering may be applied to three-dimensional problems. (Indeed it can be applied to one-dimensional problems as well.) Also, replacing the Laplacian with a more general elliptic operator does not affect the sparsity considerations, but may affect the actual convergence rate.

We close with an example of vectorized direct approach to a structured sparse problem, namely point cyclic reduction applied to a tridiagonal system.

Recall that a tridiagonal system is one in which all elements except for the three diagonals about the main diagonal are zero. A 5x5 tridiagonal matrix has the form:

(3.17)

$$\begin{pmatrix} A_1 & C_1 & 0 & 0 & 0 \\ B_2 & A_2 & C_2 & 0 & 0 \\ 0 & B_3 & A_3 & C_3 & 0 \\ 0 & 0 & B_4 & A_4 & C_4 \\ 0 & 0 & 0 & B_5 & A_5 \end{pmatrix}$$

Matrices such as this arise in many applications, especially in alternating direction implicit methods.

There are several popular methods for solving such systems on a vector processor.

While Gaussian Elimination becomes a scalar recursive algorithm due to the sparsity pattern, it can still be done efficiently for moderate size systems on the CYBER 200 by using the large register file and tuning the scalar code accordingly.

One can think of a tridiagonal system as arising from lexicographic ordering for a one-dimensional mesh. Thus the red-black ordering scheme discussed previously can be employed.

However, cyclic reduction is the most efficient method for large systems.

The basis for the cyclic reduction algorithm on tridiagonal systems is the following set of observations.

1. Consider three consecutive rows of the matrix with an even row in the middle

   2K-1 O O . . . O X X X O O O . . . O

   2K O O . . . O O X X X O O . . . O

   2K+1 O O . . . O O O X X X O . . . O

   The 2Kth row couples X_{2K} with X_{2K+1} and X_{2K-1}. If we add the proper multiples of rows (2K-1) and (2K+1) to row 2K, we can eliminate this coupling at the cost of coupling X_{2K} with X_{2K+2} and X_{2K-2}. Thus

   2K O O O X (X) X (X) X O . . .

2. If we carry out this procedure for all even rows

   (i) We have uncoupled all the even variables from the odd variables.

   (ii) The resulting system for the even variables is again tridiagonal.

   (iii) The odd equations provide a diagonal system for solving the odd variables in terms of the even variables.

   By observation (ii) we can apply cyclic reduction again and again, each time halving the size of the system remaining.

Can we vectorize this algorithm, and what kind of data storage will it take to accomplish it?

If one looks at the multipliers needed to accomplish the first stage reduction:

(3.18)
$$\left(-\frac{B_2}{A_1}\right)\ \text{row 1} + \text{row 2} + \left(-\frac{C_2}{A_3}\right)\ \text{row 3} \qquad \text{row 2}$$

$$\left(-\frac{B_4}{A_3}\right)\ \text{row 3} + \text{row 4} + \left(-\frac{C_4}{A_5}\right)\ \text{row 5} \qquad \text{row 4}$$

it is easy to see that if the original tridiagonal matrix were stored by diagonals, one could compress out the even B's and C's and the odd A's, do vector division and then apply the results to the diagonally stored odd B's and C's, resulting in the following form for the even rows:

(3.19)
$$\text{row 2} \quad 0 \quad A_2 + C_1\left(-\frac{B_2}{A_1}\right) + B_3\left(-\frac{C_2}{A_3}\right) \quad 0 \quad C_3\left(-\frac{C_2}{A_3}\right) \quad 0$$

$$\text{row 4} \quad 0 \quad B_3\left(-\frac{B_4}{A_3}\right) \quad 0 \quad A_4 + C_3\left(-\frac{B_4}{A_3}\right) + B_5\left(=\frac{C_4}{A_5}\right) 0$$

The reduction phase can be continued as long as one likes. At each stage the size of the remaining tridiagonal system will be halved.

Eventually the remaining system will be small enough to solve by Gaussian Elimination or any other technique. Then begins the substitution phase.

The following figure illustrates the order in which the variables are solved for N = 8. The variables above the line are found from the diagonal system involving the variables below the line which have already been found:

(3.20)

		1
		3
	2	5
4	6	7
8	4	2
	8	4
		6
		8

Thus at each stage of the substitution one must merge variables just solved for in order to prepare for the next level. As for the vectorizability of the diagonal system solution, if one considers the last diagonal system for N = 5,

(3.21)

$$X_1 = \frac{1}{A_1} (Y_1 - C_1 X_2)$$

$$X_3 = \frac{1}{A_3} (Y_3 - C_3 X_4 - B_3 X_2)$$

$$X_5 = \frac{1}{A_5} (Y_5 - B_5 X_4)$$

we see that the diagonal storage scheme allows this to be completely vectorized.

Experiments on the STAR 100 (6) have shown that cyclic reduction is superior to Gaussian Elimination (essentially scalar for tridiagonal systems) for $N > 150$. There is no doubt that this crossover number rises on the CYBER 203, due to its enhanced scalar performance. Tests are being conducted at the time of this writing. However, if one recalls the projected speedups in the compress and merge operations on the CYBER 203E (Specifically, see (2.27).) it is clear that the cross over point will drop dramatically on this architecture.

4. CONCLUSION

By no means have we exhausted the material that could have been presented in section 3. In particular, no mention was made concerning the problems of eigenvalue and eigenvector solution. This will be left for a later publication.

Finally, one should not forget that the CYBER 200 is a powerful scalar processor as well. An excellent example of a dual vector-scalar algorithm for Bunemann's variant of block cyclic reduction can be found in (4).

We have introduced the concept of vector processing as embodied on the CYBER 200. We have endeavored to show that the CYBER 200 hardware has the primitive operations, supported by FORTRAN, to translate many syntactic kernels into single instructions. More importantly we have endeavored to show that semantic vectorization is possible and that vector numerical analysis is being developed to provide the algorithm designer with the insights that will enable a new plateau of calculational ability to be reached.

BIBLIOGRAPHY

1. Control Data Corp. STAR 100 Hardware Manual, #60256000 Rev. 10

2. Control Data Corp. STAR 100A Hardware Manual, #60256010 Rev. 01

3. Control Data Corp. STAR FORTRAN Manual, #60386200 Rev. G

4. Kascic, M. J., A Direct Poisson Solver on STAR, Proc. of the 1978 Workshop on Vector and Parallel Processors, Los Alamos Laboratories, LA-7491-C

5. Kascic, M. J., Notes from the CYBER 200 Vector Applications Seminar

6. Madsen, N. K., Rodrigue, G. H., A comparison of Direct Methods for Tridiagonal Systems on the CDC - STAR 100, Lawrence Livermore Laboratory #UCRL-76993, Rev. 1

7. Madsen, N. K., Rodrigue, G. H., Karush, J. I., Matrix Multiplication by Diagonals on a Vector/Parallel Processor, Information Processing Letters Vol. 5, #2 June 1976

8. Nolen, J. S., Kuba, D. W., Kascic, M. J., Jr., Application of Vector Processors to the Solution of Finite Difference Equations, Fifth Symposium on Reservoir Simulation, Society of Petroleum Engineers, SPE 7675

9. Varga, R. S., Matrix Iterative Analysis, Prentice Hall 1962

10. Young, D. M., Iterative Solution of Large Linear Systems, Academic Press 1971

September, 1980

CDC CYBER 170/70/6000/200

VECTOR PROCESSING - PROBLEM OR OPPORTUNITY by Michael J. Kascic, Jr.

INTRODUCTION

The invention of the stored program computer along with the construct of high level languages has revolutionized Numerical Analysis. This revolution has spread through much of Applied Mathematics as well as some aspects of Pure Mathematics. The essence of this revolution is the capability of carrying out computations easily and efficiently which are beyond the realm of possibility of a human with pencil and paper.

For example, Kawaguti in 1953 published results for the classic Fluid Dynamics problem of two-dimensional flow past a circular cylinder based on approximately 10^5 min. of calculation using a mechanical desk calculator. Recently B. Fornberg of the California Institute of Technology has published further results on this problem for higher Reynolds number. His estimate for the problem solution on a standard commercial scalar processor of the present is 10^2 min.

It is by now widely appreciated that the difference between 10^5 min. of human resource and even 10^2 min. of computer time represents a revolution. A year and a half of a person's life is an incredible price to pay for one set of numbers. If as Hamming has said, " The object of computing is insight, not numbers", then one must consider Kawaguti's attempt a valiant failure in the art of computing.

CDC CYBER 170/70/6000/200

VECTOR PROCESSING - PROBLEM OR OPPORTUNITY (cont'd)

Although the 10^2 min. is a feasible execution time, it is not good enough. If one were experimenting with the various parameters of this problem, such as mesh size or outflow boundary conditions, even 10^2 begins to loom large. These considerations are mostly quantitative. Of far more qualitative interest is the fact that as the Reynolds number increases, the computations needed to solve even one case become enormous. In particular, up to the present, there were no reliable results for Reynolds number above 200. For all the power of scalar computers, solutions in this regime are beyond them. What we need is another revolution. That's what we've got!! Fornberg has just published computationally reliable results for the flow past cylinder problem with Reynolds number 300. Such a solution takes under 1 min. on the CDC STAR-100. This two order of magnitude increase in computational efficiency represents just as much an advance over the scalar computer as the scalar computer represents an advance over Kawaguti's heroic attempt. And yet it is only the beginning. Even at Reynolds number 300 one cannot decide between the Brodetsky vs. the Batchelor conjectures concerning the nature of such flows as the Reynolds number increases.

Where do we go from here? What is the qualitative essence of this revolution? What problems can we expect to solve with this new tool? What problems can we not expect to solve with this new tool? It is our objective here to at least develop a vocabulary so that we can ask these questions in an intelligent manner. The answers, hopefully, will keep us happily and gainfully employed for the near future.

COMPANY PRIVATE

How is the vector processor qualitatively different from the scalar processor? The scalar processor treats each individual calculation as a new adventure. Hence, within the limits of memory banking, a scalar processor can treat data that is not contiguous as well as it can data that is contiguous. Perhaps a better way to put this fact is that the scalar processor treats contiguous data as poorly as it does non-contiguous data. The vector processor, on the other hand, treats assemblages of data as units. By "taking the larger view" it is possible to gain internal parallelism and optimize the hardware within the processor.

It is not our object here to detail the internal architecture of the CYBER 200 nor to debate its merits vs. other vector architectures, nor even to demand that a system like the CYBER 200 must be 100% utilized to be cost effective. Some users will be content with attaining that portion of the system's power which is achievable with present programming practices. They will see performance improvements due primarily to the large memory and scalar processing capabilities, and secondarily to vector processing ability. Taking advantage of only a portion of a large computer's architecture is a common occurence for at least some portion of each installation's work.

Today I address those in the supercomputer community who are interested in assaulting the next plateau of computational ability. The success or failure of a given tool in this endeavor will depend as much on how intelligently it is used, as how intelligently it is designed. What we must ask is whether a given design evinces the promise of allowing human intelligence to use it to its fullest. Thus we will limit ourselves to a phenomenological description of CYBER 200 performance and how it applies to some common linear algebraic algorithms.

To quantify the performance of the CYBER 200 architecture let us introduce the following:
DEF. MEGAFLOP (abbreviated mflop henceforth): the ability to do one million floating point operations in one second.

As a scale of reference, the CDC 7600 is a 3-10 mflop computer depending upon the particular algorithm and its implementation. The key fact, however, is that this range of performance can be considered O(1) mflop. Several present vector processors, including the STAR-100 are capable of O(10) mflop. In contrast to this, consider the following hardware rates for the CYBER 205, an enhanced version of the Control Data CYBER 203. This computer can do arithmetic in both 32 and 64-bit modes and has either two or four vector pipelines.

TABLE I

CYBER 205 ADD OR MULTIPLY MEGAFLOP RATE

Column A is for the two pipe architecture and 64-bit operands.

Column B is for the two pipe architecture and 32-bit operands or the four pipe architecture and 64-bit operands.

Column C is for the four pipe architecture and 32-bit operands.

VECTOR LENGTH	A	B	C
25	20.0	22.2	23.1
50	33.3	40.0	43.7
64	38.6	47.8	54.2
100	50.0	66.7	78.4
500	83.3	142.9	220.3
1000	90.0	166.7	284.1
10000	99.0	196.1	384.3
50000	99.8	199.2	396.8
	100	200	400

It is clear that here is a tool almost orders of magnitude better than the best scalar computer of today, and almost an order of magnitude better than the present class of vector processors. And this is only the beginning. For certain algorithms, their expression as data structures leads to linked triads defined below. TABLE II details the hardware performance of such triads on the CYBER 205. {Columns A, B and C mean the same as in TABLE I.}

DEF. LINKED TRIAD - a triadic combination of two vectors and one scalar that can be evaluated as one operation.

The two most important examples of this structure are:

Vector + Scalar*Vector

(Vector+Scalar) * Vector

TABLE II

CYBER 205 LINKED TRIAD MEGAFLOP RATE

VECTOR LENGTH	A	B	C
25	25.6	27.4	28.4
50	45.5	51.3	54.8
64	54.7	63.4	68.8
100	74.1	90.9	102.6
500	149.3	238.1	339.0
1000	170.9	298.5	476.2
10000	196.7	386.8	749.1
50000	199.3	399.3	789.3
	200	400	800

To complete this description of the CYBER 205 capability, let us also mention that the full vector processing capability is sustainable while up to 16 channels are bursting at 200 megabits each. Real memory will be available with up to 4 million words. Last but not least, Control Data expects to complete the first CYBER 205 in 1980. We are thus not talking about some future dream based upon as yet unrealized breakthroughs. We are talking about a real computer.

COMPANY PRIVATE

CDC CYBER 170/70/6000/200

VECTOR PROCESSING - PROBLEM OR OPPORTUNITY (cont'd)

Indeed, if one would like to speculate about the near future, CDC is even now designing computers for the middle 80's with peak performance in the O(1000)mflop range.

What we have described so far is a calculational engine of immense raw power. Can it be harnassed to perform useful work without serious degradation? Depending upon your viewpoint, this question poses a problem or provides an opportunity.

To see this, let us perform the following thought experiment. Suppose our processor has the capability of performing calculations simultaneously in the vector and scalar processor. (The CYBER 205 has this capability). Without regard for the possible logical dependencies, given that the vector processor is one order of magnitude faster than the scalar processor, it is necessary for 90% of the calculations to be done in the vector processor in order not to be "scalar" bound. With a two orders of magnitude ratio, the percentage climbs to 99% and for three orders of magnitude 99.9%. Is this attainable? Herein lies the problem.....or the opportunity. How can we solve the problem.....or take advantage of the opportunity? Let us first outline one methodology that definitely, positively will not work.

The general philosophy of this methodology is to take all the accumulated prejudices that have been acquired working with previous generations of computers and use them to judge the possibility of attaining the performance listed in TABLES I and II.

COMPANY PRIVATE

VECTOR PROCESSING - PROBLEM OR OPPORTUNITY {cont'd}

The practical application of this philosophy is not to speak of moving to new plateaus of solving physics problems, or engineering problems, or mathematics problems, or algorithmic problems, but to speak of vectorinzing codes, usually old codes.

Once you have gotten this far, you are trapped. The only way out at this point is to deny the information theoretic counterpart of the second law of thermodynamics. As an original problem is successively transformed, it is being filtered, until it is finally a computer code. Each filter degrades the information content a little more. This is not a syntactically reversible process. Vectors may be able to be extracted from scalar code by automatic software in many cases, but consider, even if one quarter of the arithmetic operations are left to the scalar processor, and the vector processor runs at infinite rate, performance is only amplified by a factor of four. One does not have to be a professional skeptic to see that syntactic vectorization of 99% of a code is a formidable problem indeed.

Instead of thinking of all this as a problem, let's start thinking of it as an opportunity, the opportunity to solve problems beyond the range of the scalar and vector processors of today. Recently a researcher in fluid dynamics was discussing an algorithm he would have liked to pursue. He discarded it out of hand because it would have involved solving 1000 equations in 1000 unknowns. This was obviously unthinkable to him. Using Gaussian Elimination {without pivoting}, this set of equations can be solved in under 2.5 seconds on the four pipe CYBER 205 When advised of this fact, he got that far away look in his eyes. What new avenues of thought does this processor open up?

COMPANY PRIVATE

CDC CYBER 170/70/6000/200

VECTOR PROCESSING - PROBLEM OR OPPORTUNITY {cont'd}

Well, don't run off to the ivory tower. This is not a case of unconstrained optimization. Let's try to outline a positive approach to this opportunity that satisfies the necessary constraints.

First, it is clear that not every problem should be solved on a super vector processor. Whether or not the large real memory makes even the scalar component worth the price of the whole computer is not the point. We regularly solve problems with hand or desktop instruments that once were mainframe problems. Why do we still have mainframes? Obviously because we have enlarged our problem horizon as we have enlarged our computing capability. Let's not stop now.

This does not mean, of course, that the large problems in today's terms cannot be computed efficiently on machines of the CYBER 205 class. Indeed, a major component of the viability of a new computer in the marketplace is its ability to give a quantum performance gain with purely syntactic transition. But when you deal with a machine with a peak stream rate of 800 mflop, 80 mflop just doesn't look all that impressive any more.

What have we concluded so far:

1. Syntactic conversion of codes will gain us respectable performance but will not do in the long run.
2. Components of existing codes that can take advantage of the super vector processor's stream rate should be identified to aid in the transition.
3. New programs should be developed in terms of algorithms that map well onto the super vector processor.

various applicable disciplines. While the more subtle results of this activity are yet to be determined, a good place to start is with

VECTOR NUMERICAL LINEAR ALGEBRA

The algorithms of Linear Algebra provide a convenient starting point for constructing our desired algorithm class. Matrix multiplication and linear equation solution are not only important in real world applications, but they have also achieved a certain status as benchmark problems. {In some ways, our analysis is paralleled by that of {3} in the construction of the BLAS. However, we differ in that in terms of our algorithm criteria, the BLAS are too low level, too general {and yet too specific}, and too syntax oriented. They may be fine on a scalar processor, but will not do on the super vector processor}.

We shall limit ourselves here to brief examples of several elementary operations and how present knowledge of their proper implementation on the CYBER 205 does or does not fit our criteria.

Let us first establish a vocabulary of matrix sparsity. A matrix is considered full if a priori all elements must be considered non-zero. A matrix is considered large banded iff all elements outside the diagonals P diagonals from the main diagonal are zero and all elements inside these diagonals are considered non-zero. A matrix is structured sparse iff the non-zeros lie along a fixed relatively small number of diagonals. {There is a fourth class which consists of those matrices that have O(N) non-zeros in a completely random pattern. We shall not consider these}.

These definitions are compromises, of course. Many matrices occuring in practice can fit either into the large banded class at the cost of ignoring a number of zeros, or into the structured sparse class if blocks are considered instead of individual values. Even in scalar case (see [1]) it has been recognized that there are competing factors of performance evaluation. However, in the case of the super vector processor there is a whole gamut of new considerations, not the least of which is criterion # 5, i.e., the monotonic character of vector performance.

We list below the expected performance of the CYBER 205 (in terms of mean vector length) on certain fundamental linear algebraic algorithms. For more details and further references, see [2]. The terms "row", "col." and "diagonal" refer to the storage order of a given matrix.

TABLE III

OPERATION	DATA STRUCTURE	MEAN VECTOR LENGTH	LINKED TRIAD
MATRIX MULTIPLICATION			
C=AB FULL	A col. A--- B--- or B row C col. C row	N	YES
LARGE BANDED	Ditto	2P	YES
STRUCTURED SPARSE	A diagonal B diagonal C diagonal	lengths of diagonals	NO
GAUSSIAN ELIMINATION			
AX=Y			

COMPANY PRIVATE

PROCESSING - PROBLEM OR OPPORTUNITY {cont'd}

REDUCTION			
FULL	A row or col.	2N/3	YES
LARGE BANDED	A row or col.	P	YES
SUBSTITUTION			
FULL	A row	N/2	INNER PRODUCT
	A col.	N/2	YES
LARGE BANDED	A col.	P	YES
	A row	P	INNER PRODUCT
FULL GAUSSIAN			
INVERSION			
AX=Y			
REDUCTION	A row or col.	2N/3	YES
	AY augmented rows	5N/3	YES
SUBSTITUTION	A row	N	YES
	A col.	N/2	YES

SOR on full or large banded systems uses the matrix multiplication and substitution algorithms listed above.

STRUCTURED SPARSE {SOR} as applied to an N^3 mesh to solve elliptic BV problem	red-black node order	$N^3/2$	NO

COMPANY PRIVATE

CYCLIC REDUCTION FOR TRIDIAGONAL SYSTEM

AX=Y	A diagonal	N/log N	NO

Let us examine the algorithms in TABLE III in the light of the five criteria. First, they are general purpose. Many "dusty deck" programs could benefit immediately from insertion of these algorithms. As for specificity, admittedly this is an area where much growth is needed. This will not happen until the aforementioned loop is closed to connect computational entities with the original mathematical ones. This is the aim of the Control Data sponsored research in this area. While these algorithms are not considered low level from the programming standpoint, from the applications problem viewpoint they certainly are. On the other hand, they are high level enough to allow a super vector processor like the CYBER 205 not to get bogged down, i.e., to allow the actual performance to closely match the theoretical performance for a given vector length. Finally, within the limitation of real memory, each of these algorithms is monotonically more efficient as the problem size increases.

In conclusion, the super vector processor presents an unprecedented opportunity to move into new calculational regimes, provided we approach it intelligently. Part of this intelligence is the recognition that such a processor must be seen as a full partner in the quest for insight along with physical intuition, engineering ingenuity and mathematical rigor.

COMPANY PRIVATE